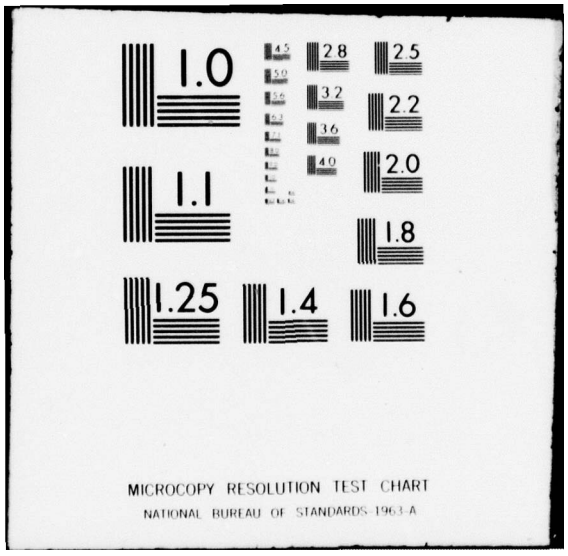
AD-A060 668 DIGITAL TECHNOLOGY INC CHAMPAIGN IL F/G 9/2
INFE PROGRAM DESIGN SPECIFICATIONS. PHASE B. NFE SOFTWARE PROGR--ETC(U)
MAY 78 S F HOLMGREN, D C HEALY, D A WILLCOX
UNCLASSIFIED DTI-6 SBIE-AD-E100 082 NL

1 OF 2
AD
A060688


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Digital Technology Incorporated

LEVEL II

(1)

INFE
Program Design Specifications

Steven F. Holmgren
David C. Healy
David A. Willcox

May 15, 1978
Document 6

(14)

DTI ~~Document Number~~-6

(6)

INFE PROGRAM DESIGN SPECIFICATIONS
Phase B, NFE Software Program Design Specifications

by

(10) Steven F. Holmgren
David C. Healy
David A. Willcox

Prepared for the
Defense Communications Agency
under ~~contract~~
(15) DCA 100-77-C-0069
Phase B Network ~~Frontend Research~~ and Development

by

(18) SBIE (19) AD-E100 082

DIGITAL TECHNOLOGY INCORPORATED
Champaign, Illinois 61820

(11) 15 May 1978

(12) 115 p.

Approved for Release: Peter A. Alsberg
Peter A. Alsberg, President

410 796

B

ABSTRACT

This document contains program design specifications for the Interim Network Front End (INFE) under development by Digital Technology Incorporated. The INFE will connect a WWMCCS H6000 to AUTODIN II. The INFE is an modification of the Experimental Network Front End (ENFE). The ENFE connected a WWMCCS H6000 to the ARPANET. The INFE and the ENFE differ only in the network protocols that they implement. Only AUTODIN II protocol software is described here. Detailed descriptions of the Transmission Control Protocol (TCP) and Terminal to Host Protocol (THP) modules follow.

TABLE OF CONTENTS

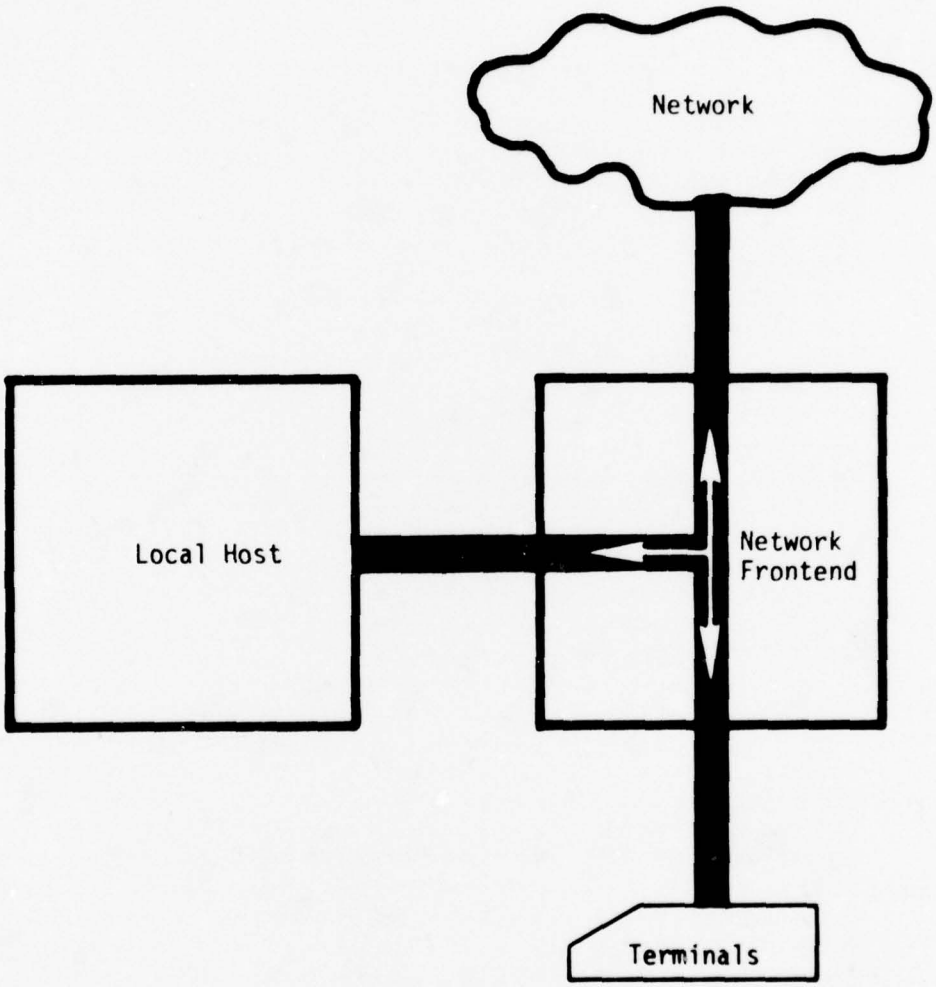


Figure 1: Place of the Network Frontend

BACKGROUND

A Network Front End (NFE) is a computer system interposed between a host computer and a network. The NFE relieves the host of the burden of network interface software. It also provides an alternative mechanism for local terminals to access the network when the host is down. The place of the Network Front End is shown in Figure 1.

The software specified here will implement a WWMCCS Interim NFE (INFE) that will provide initial operational capability (IOC) for AUTODIN II. This is Phase b of the WWMCCS NFE program. Phase B was preceded by the development of Network UNIX and a Phase A Experimental NFE (ENFE).

UNIX is a general purpose operating system developed by Bell Telephone Laboratories for PDP-11 computers. In 1975, University of Illinois staff added ARPANET Network Control Program (NCP) software and ARPANET terminal (Telnet) software to UNIX to make Network UNIX.

In 1976, the Network UNIX staff added Host to Front End Protocol (HFP) software to Network UNIX creating the Phase A WWMCCS Experimental NFE (ENFE).

In October 1976, the Network UNIX and ENFE development team formed Digital Technology Incorporated (DTI). DTI is developing an INFE by adding AUTODIN II protocol to the ENFE.

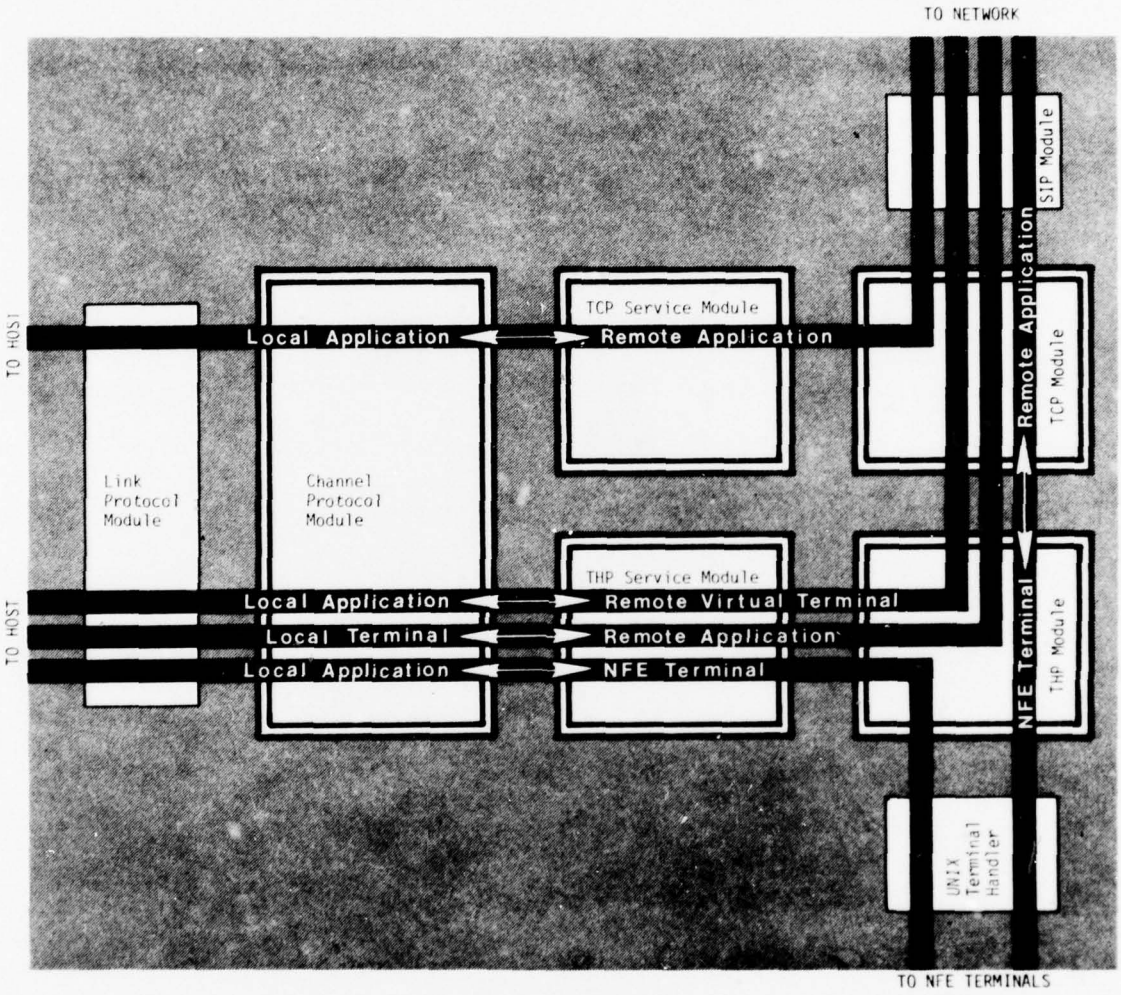


Figure 2: INFE Routes and Modules

INFE ARCHITECTURE

The INFE facilitates data transfer between

a. a host and AUTODIN II,
b. a host and INFE terminals, and
c. INFE terminals and AUTODIN II.

Protocols

HFP. Services in the INFE support processes in the host. The processes access their services using Host to Front End Protocol (HFP). HFP has three levels (link, channel, service). The lowest level (link level) controls the hardware interface between the host and front end. The next level (channel level) implements logical channels between all processes in the host and all services in the INFE. Each channel is sequenced and flow controlled. HFP Messages are exchanged on HFP channels. HFP Messages have two major parts: a header and a text field. The structure of the header is defined by the channel level of HFP. The structure of the text field is defined by the third level of HFP (process-service level). There is a separate process-service level HFP specification for each service.

TCP. Transmission Control Protocol (TCP) is a process-to-process data transfer protocol. TCP is used on AUTODIN II to insure the reliable and error free transmission of data. TCP transfers data segments over AUTODIN II using a Segment Interface Protocol (SIP). Processes use TCP letters to communicate via AUTODIN II. TCP error controls, flow controls, and sequences TCP letters. A transmitting TCP breaks letters into segments for SIP transmission over AUTODIN II. SIP handles the AUTODIN II

hardware interface. A receiving SIP passes segments onto its TCP partner for reassembly into letters.

SIP. The INFE will be available before AUTODIN II IOC. Initially the SIP Module implemented on the INFE will be a Pseudo Segment Interface Protocol (PSIP). The PSIP will appear to be a SIP Module to the TCP Module. However, the PSIP will use the ARPANET, not AUTODIN II, for communication. When AUTODIN II becomes available, a SIP will be substituted for the PSIP.

THP. Terminal to Host Protocol (THP) defines the terminal communication standards for AUTODIN II. THP uses TCP as its data transport mechanism.

Software Modules.

Figure 2 shows the communication routes supported by the INFE and the software modules implemented in the INFE. The SIP Module (PSIP Module) and the UNIX Terminal Handler are implemented as UNIX device drivers. All other INFE modules are implemented as separate UNIX user-level processes. We will sometimes use the terms process and device driver when discussing these modules.

Link Protocol Module. The INFE Link Protocol Module implements the link level protocol of HFP. This module will be an identical copy of the ENFE Link Protocol Module. The Phase A ENFE experiments showed that the link level protocol should be augmented with flow control and error detection/correction. These facilities could be added in the future.

Channel Protocol Module. The INFE Channel Protocol Module implements the channel level protocol of HFP. The ENFE Channel Protocol Module will be used with minor changes. These changes correspond to the minor change made in HFP. (The HFP signal command has been eliminated.)

TCP Service Module. The INFE TCP Service Module implements the process-service level of HFP for TCP service. This module maps host process requests for TCP into standard UNIX I/O calls (open, close,read, etc.) to the TCP Module. The INFE TCP Service Module will use the structure of the ENFE Host-Host Service Module.

THP Service Module. The INFE THP Service Module implements the process-service level of HFP for THP service. HFP Messages containing THP requests are mapped into standard UNIX I/O calls to the THP Module. The INFE THP Service Module will use the structure of the ENFE Server Virtual Terminal Service Module.

TCP Module. The INFE TCP Module implements TCP. This will be a major new module. Is is discussed in detail below.

THP Module. The INFE THP Module implements THP. This also will be a major new module. It is discussed in detail below.

SIP Module. The INFE SIP Module will implement SIP on the ARPANET. That is, it will be a PSIP that will mimic an AUTODIN II network device driver. The ENFE ARPANET device driver will not be used in the PSIP.

UNIX Terminal Handler. The INFE UNIX Terminal Handler controls terminals attached to the INFE. The ENFE UNIX Terminal Handler will be modified to take advantage of non-blocking terminal I/O.

TCP MODULE

Introduction

The TCP Module will implement the latest version of TCP ["Internetwork Transmission Control Protocol", TCP Version 3, Cerf and Postel, January 1978]. It is expected that this version will be very similar to the final AUTODIN II TCP specification. The connection access control mechanisms described in the preliminary TCP specification ["Transmission Control Protocol Specification", Postel, Garlick, and Rom, July 15, 1976] will be implemented by the TCP Module. When a final AUTODIN II TCP specification is available, the TCP Module will be modified to match.

The ENFE software requires a PDP-11/45 or PDP-11/70. Smaller PDP-11's (e.g., 11/34 and 11/40) do not have adequate kernel address space. Yet there is strong interest in using smaller PDP-11's as INFE's. Reduction of ENFE kernel size is required.

In the ENFE, the NCP is split into a kernel portion and a user portion (called the NCP Daemon). For maximum throughput, the INFE TCP should also have a kernel portion. However, the size of the kernel portion would preclude operation on smaller PDP-11's. Thus TCP is being implemented as a single user process outside of the kernel address space.

User level processes (i.e., most of the INFE software modules) communicate with the TCP process (and each other) via standard UNIX I/O calls (open, close, read, etc.). To do this,

we added a standard UNIX I/O interface to the interprocess communication facility we developed while at the University of Illinois. I/O calls (e.g., read, write, open) directed to the TCP process are changed into I/O messages. These messages contain the message type (open, close, read, etc.), identify the sender (process id, user id, file id), and identify the sender's buffer. New UNIX calls were implemented to allow a process to:

wait for an I/O message,

reply to an I/O message, and

access the user's buffer.

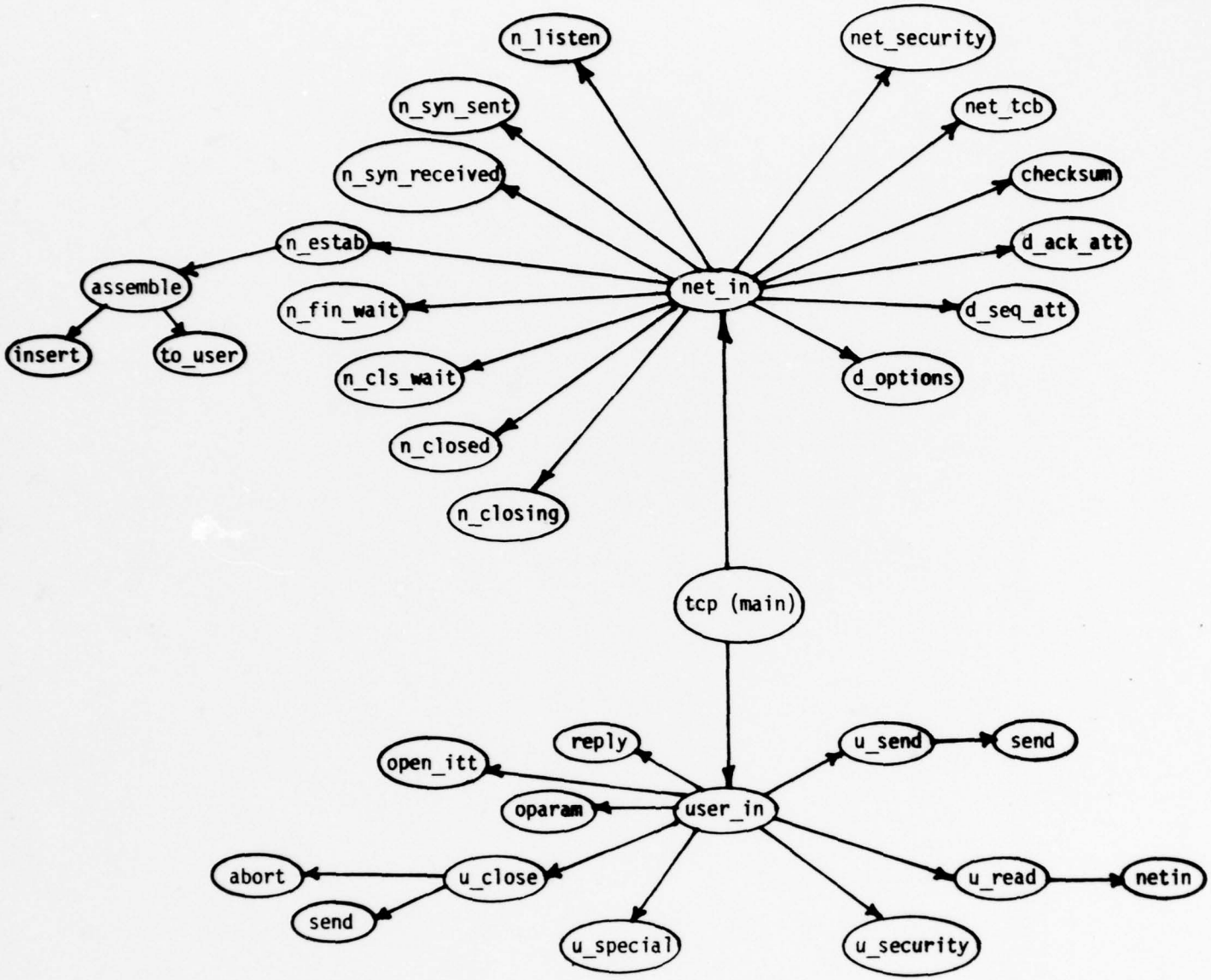
I/O requests are received from the THP process, the TCP process, and other INFE processes. I/O replies are received from the Pseudo Segment Interface Protocol (PSIP) device driver when network reads and writes have been completed.

Non-blocking read and writes are issued to the PSIP device driver. I/O replies are sent to the requesting process when I/O requests have been processed.

Data received from a user program is packaged into segments for transmission over the network. Each segment is assigned a sequence number and checksummed. The sequence numbers are used for flow control and for duplicate and missing segment detection. Checksums are used to insure data integrity.

When a correctly sequenced and checksummed segment is received from the network, a network acknowledgement is transmitted. Received segments are assembled into letters. Complete letters are passed to user programs. Unacknowledged

segments are retransmitted on timeout (possibly generating duplicate segments).



TCP SUBROUTINE CALLING HIERARCHY

Control Flow

The TCP Module is a loop (MAIN) which waits for and then processes UNIX I/O messages. Each message is associated with a new or existing network connection. When an I/O message is received, the TCP Module locates or constructs a Transmission Control Block (TCB) for the connection. The connection state (estab, syn sent, etc) and message type (open, close, read, etc.) are used to call a function specific subroutine. The TCP Module is divided in two major sections: network and user. The network section handles I/O replies from the PSIP device driver and sends I/O replies to user-level processes (e.g., TCP Service Module, THP Module). The user section handles I/O requests from user-level processes.

Network. NETIN is called when an I/O reply is received from the PSIP device. NETIN calls CHECKSUM to validate segment data integrity. NETIN calls NET_TCB to locate the wTransmission Control Block for the connection. NET_SECURITY is called to implement AUTODIN II security, precedence, and transmission control code handling. The segment header is processed (D_OPTIONS, D_SEQ_ATT, and D_ACK_ATT) and a connection state subroutine is called. The connection state subroutines are:

N_CLOSE,
N_LISTEN,
N_SYN SENT,
N_SYN_RCVD,
N_ESTAB,
N_FIN_WAIT,
N_CLS WT, and
N_CLOSING.

These implement the TCP connection open, synchronization, and

close mechanisms. N_ESTAB processes user data. N_ESTAB uses segment sequence and acknowledgement attributes (i.e., acknowledgement in sequence and believable) to perform initial processing. N_ESTAB calls ASSEMBLE to handle state dependent processing of segments. ASSEMBLE calls INSERT to combine network segments into letters. ASSEMBLE then calls TO_USER to pass I/O replies for any complete letters to the user.

User. USER_IN is called when an I/O request is received from a user. USER_IN checks the request for an open function. If an open request is found, OPARAM is called to parse the parameters. OPEN_ITT is then called to perform the open. If the request in not an open request, U_SECURITY is called to enforce AUTODIN II security, precedence, and transmission control code conventions. The request function type is then used to call one of four subroutines: U_CLOSE, U_READ, U_SEND, or U_SPECIAL. Each of the subroutines uses the connection state to determine processing.

U_CLOSE checks the connection state. If the connection is partially established, ABORT is called. If the connection is established, a TCP FIN segment is generated, and SEND is called to initiate its transfer to the network.

U_READ checks the connection state. If the connection is not established, an error I/O read reply is sent to the user. If the connection is established, a descriptor for the read request buffer is linked into the t_rcvq queue. U_READ then calls NETIN to copy any previously received network data to the user buffer.

U_SEND checks the connection state. If the connection is not established, an error I/O write reply is sent to the user. If the connection is established, a descriptor for the write buffer is linked into the t_sndq queue. SEND is then called to initiate the transfer of user data to the network.

U_SPECIAL is called to handle connection maintenance functions. At present, only a status function is defined. Other functions will be defined as needed. The status function returns connection status information. The connection state, local and remote ports, receive and transmit flow control information, and security information are returned to the user.

Data Flow

Data flows in two directions: from a user-level process to the network, and from the network to a user-level process.

Network-to-user. The t_rcvq is used to store user-level read requests. Each entry in this queue contains a descriptor which defines a user-level buffer in which network data is to be stored. Network data is processed by NETIN. Each user buffer is filled with network data. When a buffer is full, a read I/O reply is sent to the user-level-process. This is repeated until a network segment is emptied or the t_rcvq queue is exhausted.

User-to-Network. User-level data is processed by U_SEND. Each write request contains a descriptor for data to be sent over a connection. The descriptors are linked into a fifo ordered queue (t_sndq) by SEND. Each entry in the queue is processed. Each entry contains user-level data to be sent to a remote

subscriber. When network acknowledgements are received, entries are deleted from the queue and write I/O replies are sent to the user-level-process. If a retransmission is required, the queue is reprocessed and SEND is again called to transfer queue entries to the network.

TCP DATA STRUCTURES

This is a description of the format of a TCP Module Transmission Control Block (TCB). A TCB is allocated for each network connection. The first column specifies the field size. Int defines a 16-bit field (e.g. int a;) or an array of 16-bit fields (e.g. int a[3]); Char defines an 8-bit field (e.g. char a;) or array of 8-bit fields (e.g. char a[3]). Struct defines a complex data structure (e.g., struct {char name[30]; int SSN[3]} person). "Struct person *abc" defines abc to a a 16-bit pointer to a struct with the person format.

```
struct tcb {
        int     t_lport[2];     /* local port */
        int     t_fport[2];     /* foreign port */
        int     t_ftcp[2];      /* foreign tcp and net */
        int     t_rwin;         /* receive window */
        int     t_swin;         /* send window */
        int     t_out;          /* num. bytes in transmission */
        int     t_send;         /* num. bytes waiting to send */
        int     t_acked;        /* bytes acked not pass to usr */
        int     t_s_seq[2];     /* last acked by fgn tcp */
        int     t_s_nxt[2];     /* next seq we should use */
        int     t_r_seq[2];     /* last seq we acked */
        int     t_syn_seq[2];   /* seq num consumed by the SYN */
        int     t_fin_seq[2];   /* seq num consumed by the FIN */
        int     t_s_urg[2];     /* send urgent seq. number */
        int     t_r_urg[2];     /* rcv urgent seq. number */
        int     t_s_buf;        /* send buf size  */
        int     t_r_buf;        /* receive buf size  */
        int     t_s_frg;        /* curr send buff fragment */
        int     t_r_frg;        /* current rcv buffer fragment */
        int     t_state;        /* state -- 0 to 7 */
        int     t_ntime;        /* number of timeouts */
        int     t_nrcv;         /* user reads outstanding */
        int     t_nsnd;         /* user writes outstanding */
        int     t_prec;         /* connection precedence */
        int     t_s_sec;        /* send security */
        int     t_r_sec;        /* receive security */
        int     t_tcc;          /* transmission control code */
        int     t_flags;        /* flags */
        int     t_ctrl;         /* control bits to send */
        char    *t_pkt;         /* curr. read bfr */
        int     t_nopen;        /* number times opened */
        int     t_closeid;      /* reply id for q'd close */
        struct rcvq *t_rcvq;    /* users rcv descriptors */
```

```
        struct sndq *t_sndq;    /* users snd descriptors */
        struct idq *t_opnq;     /* open reply descriptors */

};
```

The format of the internet header follows. The internet header is found at the beginning of each network packet. It contains the source address, destination address, and packet length (in octets).

```
struct inpacket
{
        char    in_prot;        /* version and protocol */
        char    in_opt;         /* OPT, DF, TOS */
        int     in_length;      /* len of packet in octets */
        int     in_pkid;        /* internet packet id */
        int     in_fr_num;      /* fragment number */
        int     in_dtcp[2];     /* dest tcp and net id */
        int     in_dport[2];    /* destination port */
        int     in_stcp[2];     /* source tcp and net id */
        int     in_sport[2];    /* source port */
        char    in_opts[];      /* options if any */
};
```

The format of a TCP segment header follows. A TCP segment header immediately follows each internet header. The header contains the information required to send and receive data and control information. These fields include send and acknowledgement sequence numbers, receive window, checksum, and an urgency ptr.

```
struct packet
{
        int     p_seq[2];       /* seq number of this segment */
        int     p_ack[2];       /* what we are acking */
        int     p_ctrl;         /* tcp control bits */
        int     p_window;       /* window we are given wrt n_seq */
        int     p_cksum;        /* checksum for segment */
        int     p_urg;          /* urgent ptr */
        char    p_opts[];       /* if any */
};
```

TCP SUBROUTINES

The following is a list of the the major TCP Module subroutines with a brief description of each subroutine function.

ack
see what has been acknowledged by the remote TCP

assemble
handle network data reassembly

checksum
checksum a network packet

d_ack_att
determine ack validity

d_seq_att
determine sequence number validity

d_options
process the options in the packet

init
initialize queues and open log and PSIP files

main
driving subroutine

netin
high level process of network packet

net_security
security check of a network packet

n_closed
process segment when TCP is in the closed state

n_closing
process segment when TCP is in closing state

n_cls_wt
process segment when TCP is in close wait state

n_estab
process segment when TCP is in established state

n_fin_wait
process segment when TCP is in fin wait state

n_listen
process segment when TCP is in listen state

n_syn_rcvd

process segment when TCP is in sync received state

n_syn_sent
process segment when TCP is in syn sent state

oparam
process open request parameters

reset
reset the connection

send
send data to the network

to_net
write the segment to the network

to_user
write the segment to the network

user_in
first level handling of user request

u_abort
process a users abort request

u_close
process a users close request

u_read
process a users read request

u_send
process a users write request

THP MODULE

Introduction

Terminal-to-Host Protocol (THP) is a virtual terminal protocol which uses TCP as a data transport mechanism. The THP Module provides a mechanism which allows INFE terminals to access the host, and implements the THP described in "Terminal-to-Host Protocol Specification", [Postel, Garlick, and Rom, July 15, 1976]. Is is expected that this version will be very similar to the final AUTODIN II THP specification. When a final AUTODIN II THP specification is available, the THP Module will be modified to match.

Data Flow

Inputs. The THP Module receives I/O messages from the THP Service Module, INFE terminals (UNIX Terminal Handler), and the TCP Module. The THP Module receives I/O requests from the THP Service Module. These requests are generated in response to THP Service Module I/O calls (open, close, read, write, and iospcl). The THP Module receives I/O replies from the INFE terminal device driver (UNIX Terminal Handler) and the TCP Module. These replies are generated in response to outputs described below.

Outputs. The THP Module sends I/O messages to the THP Service Module, INFE terminals, and the TCP Module. The THP Module issues I/O calls to the UNIX Terminal Handler and the TCP Module. The THP Module sends replies to the THP Service Module.

Data is received from the TCP Module, INFE terminals, and the THP Service Module.

TCP Module Data. I/O replies are received from the TCP Module when previously issued UNIX I/O reads have been completed. When a read I/O reply is received, the buffer associated with the read has been filled with network data. The data is broken into one or more pieces which are queued on the to_user queue.

The to_user queue is processed by the TO_USER subroutine. The queue may contain user data and/or THP controls. Data is copied into a per-connection line buffer and removed from the queue. THP option negotiations for INFE terminals are processed immediately by NET_OPTION. Other THP controls are handled by special routines, one per control type. If the connection is associated with an INFE terminal, data is transferred to the terminal from the line buffer. If the connection is associated with the THP Service Module, the data is transferred to buffers received in previous I/O read requests.

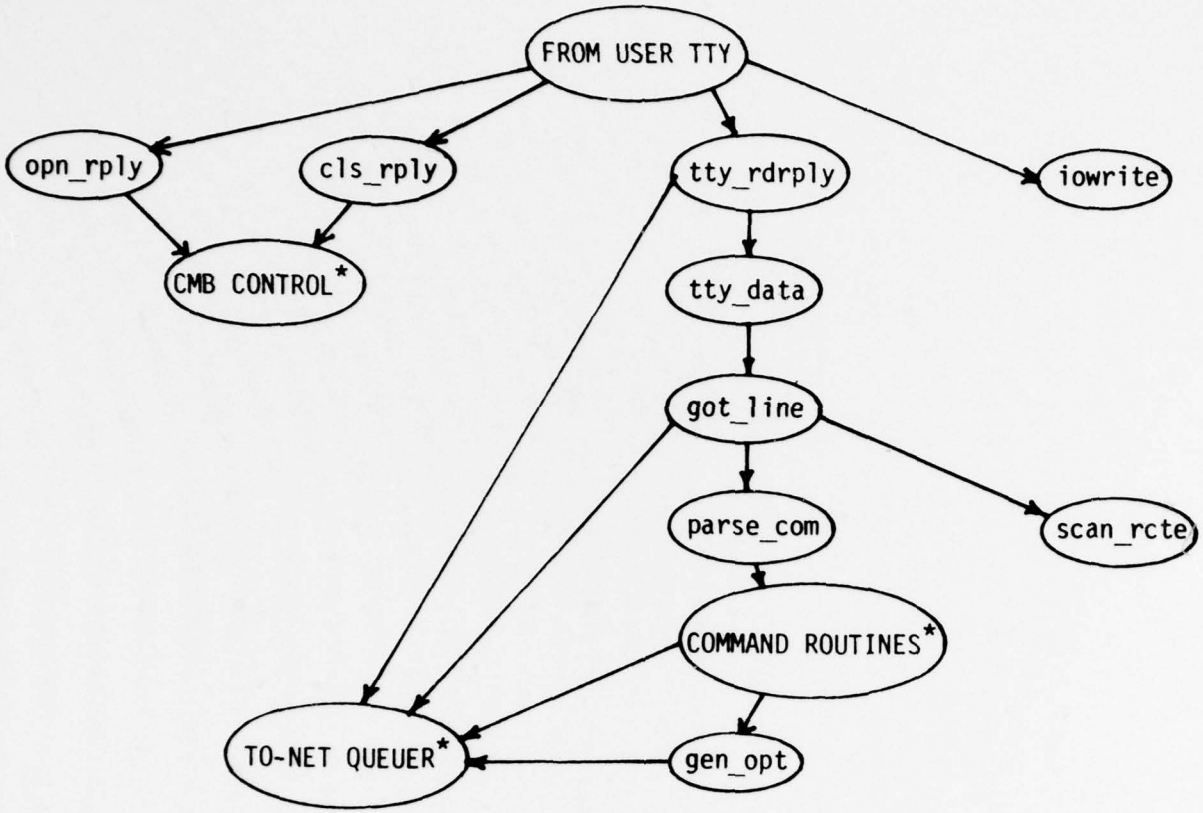
Terminal and THP Service Module Data. When data is received from an INFE terminal or the THP Service Module, it is scanned for locally-defined commands or special characters (e.g. line feed). Local commands or special characters are processed by the PARSE_COM subroutine. THP controls and blocks of data are queued on the to_net queue.

The to_net queue is processed by the TO_NET subroutine. The queue may contain data or THP controls to be sent over the net. Data and controls are copied into network transmit buffers

as the buffers become available. If data was received in a THP Service Module write I/O request, a write IO reply is sent to the THP Service Module when all of the data has been copied into network transmit buffers.

Control Flow

The THP Module MAIN subroutine is a loop which waits for and initiates processing of I/O messages. The source of the message is used to branch to one of three sections of software: From User TTY software, From TCP Module software, or From THP Service software. When control returns from one of the three sections, the to_user queue is examined for any data to be sent to the user. If there is data, the TO_USER subroutine is called. The to_net queue is then examined for data to be sent to the TCP Module. If there is data, the TO_NET subroutine is called.



THP SUBROUTINE CALLING HIERARCHY FROM USER TTY

* subroutine groups

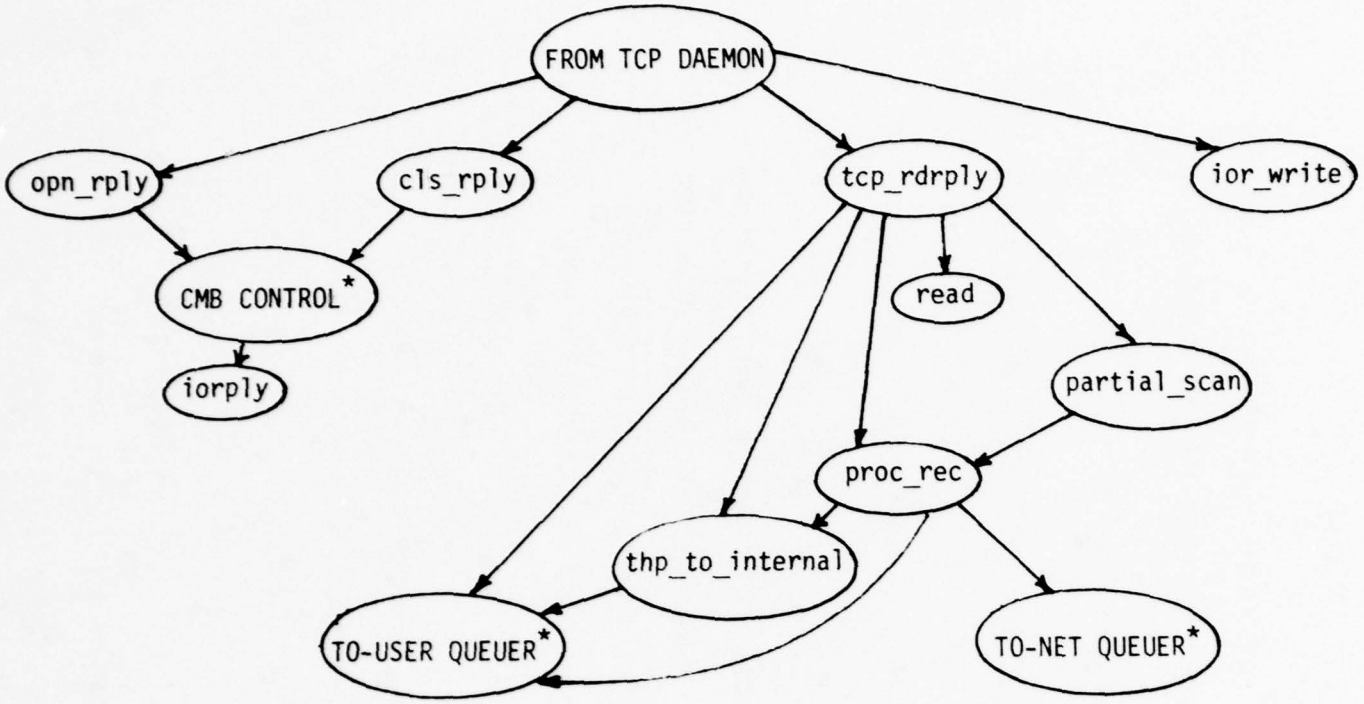
From User TTY. The From User TTY software is called when an I/O reply is received from an INFE terminal. The reply type is used to call one of four major subroutines: OPN_RPLY, CLS_RPLY, TTY_RDRPLY, or IOR_WRITE.

OPN_RPLY is called when an I/O open reply is received. Receipt of an open reply indicates that a terminal has become active. OPN_RPLY allocates input and output line buffers and issues a non_blocking read to the terminal.

CLS_RPLY is called when an I/O close reply is received. CLS_RPLY initiates the removal of the Connection Management Block (CMB). When the network connection is closed, the CMB is deallocated.

TTY_RDRPLY is called when an I/O read reply is received. TTY_RDRPLY calls TTY_DATA to process the data, performs general buffer housekeeping, and issues another non-blocking read call to the terminal. TTY_DATA scans through the data looking for special characters or THP controls. When a complete input line is received (line feed), GOT_LINE is called. GOT_LINE calls PARSE_COM if the line is a local THP Module command(e.g., open a connection, close a connection, etc.). PARSE_COM calls a command specific subroutine. Some of the command subroutines call GEN_OPT or Q_CTRL to put THP controls into the to_net queue. GOT_LINE calls SCAN_RCTE if any data is to be sent to the network. SCAN_RCTE implements the Remote Control Transmission and Echoing THP control. GOT_LINE then calls a general purpose subroutine to link the data into the to_net queue. The to_net queue will be processed by the TO_NET subroutine.

IOR_WRITE is called when a write is received from an INFE terminal. It deletes the written characters from the terminal write buffer.



THP SUBROUTINE CALLING HIERARCHY FROM TCP DAEMON

* subroutine groups

From TCP Module. The From TCP Module software is called when an I/O reply is received from the TCP Module. The reply type is used to call one of five subroutines: OPN_RPLY, CLS_RPLY, IOR_WRITE, TCP_RDRPLY, or IOR_SPCL.

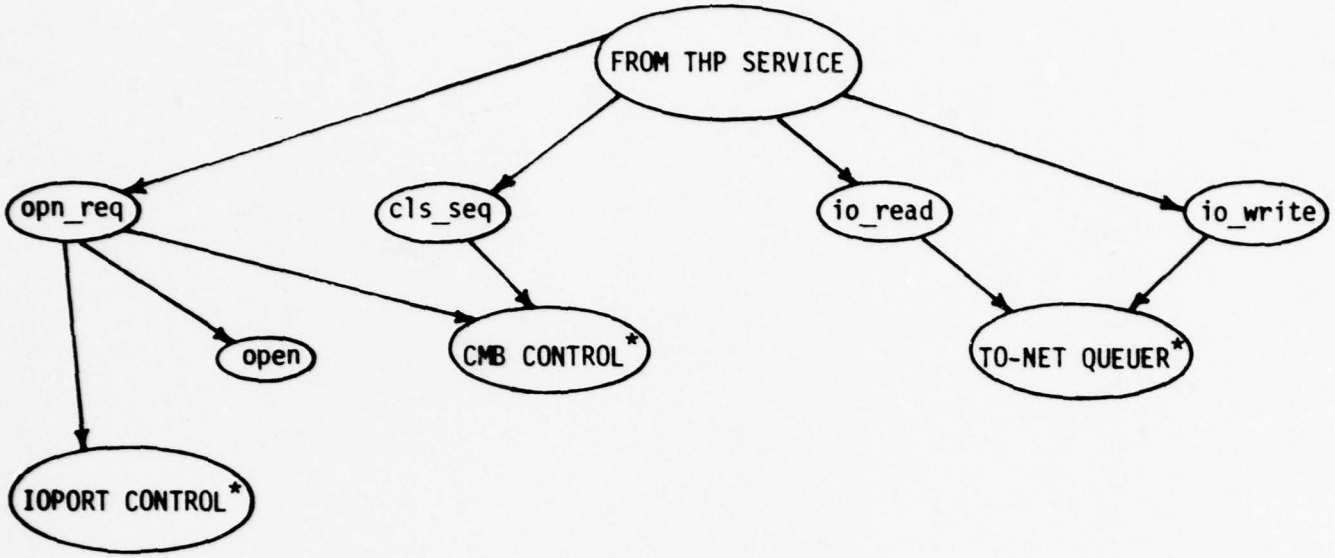
OPN_RPLY allocates network receive and transmit buffers and issues non-blocking read calls to the TCP Module.

CLS_RPLY is described above.

The IOR_WRITE subroutine is called when a write I/O reply is received from the TCP Module. If the write was unsuccessful, the connection is marked as closing. The write buffer is put back into a free pool.

TCP_RDRPLY is called when a read I/O reply is received from the TCP Module. If a partial THP header has previously been received, PARTIAL_SCAN is called. PARTIAL_SCAN accumulates characters until a complete THP header is received. When a full header is received, PROC_REC is called. If there is a full THP header and a partial THP data record previously received, TCP_RDRPLY calls THP_TO_INTERNAL to copy complete output lines to the to_user queue. If a full THP record has been received, PROC_REC is called. PROC_REC uses the record type to determine specific processing. If the record is a data record, THP_TO_INTERNAL is called. If the record is a THP control, the specific control is processed. If the control requires a network response, the response is created and queued on the to_net queue. If the control requires user processing, a control message is created and queued on the to_user queue.

The IOR_SPCL subroutine is called when a special I/O reply is received from the TCP Module. If the special request was issued in response to a request from the THP Service Module, an I/O special reply is sent to the THP Service Module. If the reply indicates that an URGENT indication has been received, a flag is set to indicate that the connection should be put into read fast mode. The connection will stay in read fast mode until a end of urgent indication is received in a read reply.



THP SUBROUTINE CALLINING HIERARCHY FROM THP SERVICE

* subroutine groups

From THP Service Module. The From THP Service software is called when an I/O request is received from the THP Service Module. The request type is used to call one of five major subroutines: OPN_REQ, CLS_REQ, IO_READ, IO_WRITE or IO_SPCL.

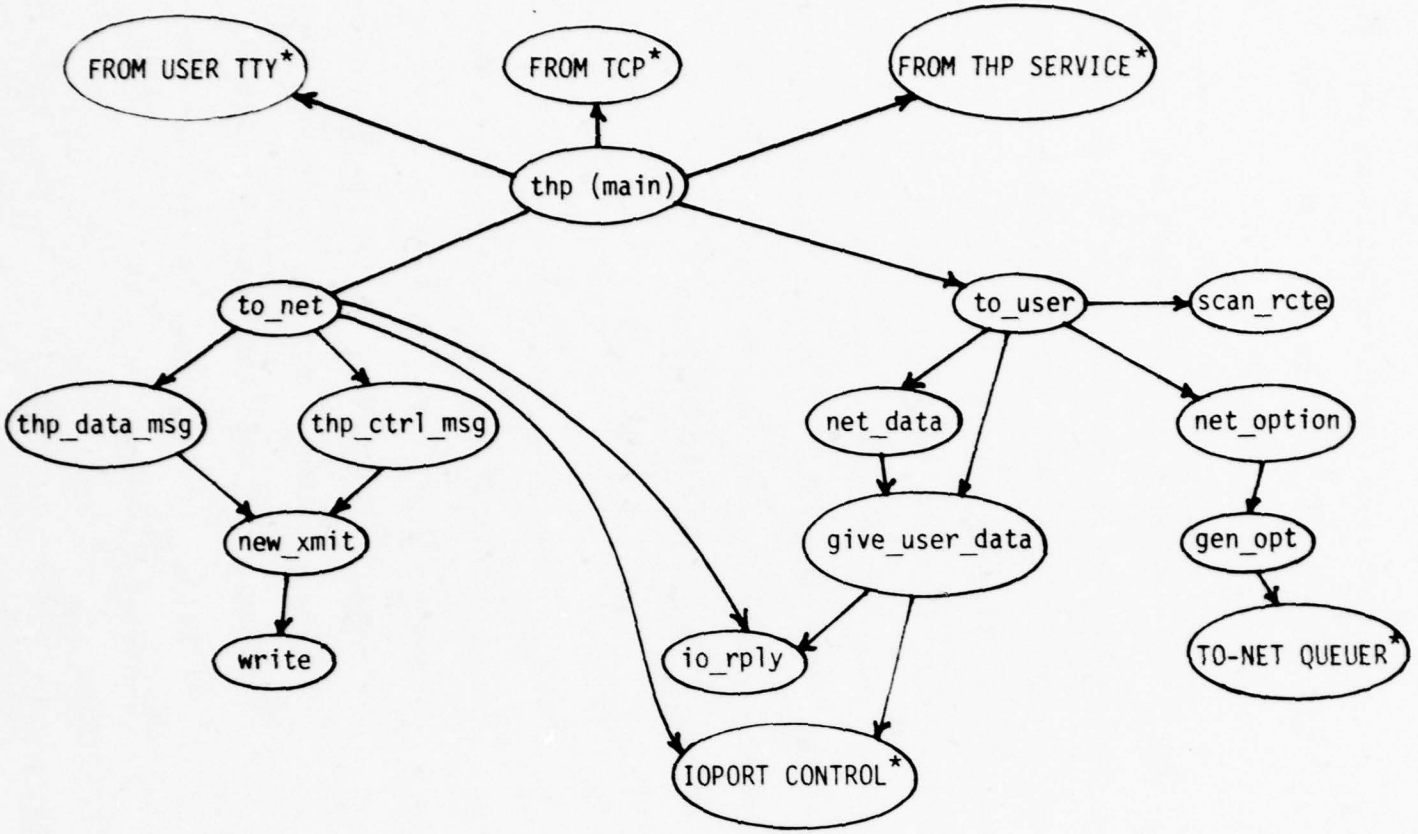
The OPN_REQ subroutine is called when an I/O open request is received. OPN_REQ tries to allocate a Connection Management Block (CMB) for the connection. If there are no CMB's available, an I/O reply with an error status is returned to the THP Service Module. If a CMB was available, it is initialized. The open request is linked on the to_net queue to be sent to the TCP Module.

CLS_REQ is called when a close I/O request is received. CLS_REQ checks the connection state. If it is already closed, the close request is sent back to the THP Service Module with an error status. Otherwise, the close request is linked on the to_net queue to be passed on to the TCP Module.

IO_READ is called when a read I/O request is received. If the connection is closed, or if the number of allowable reads has been exceeded, an I/O reply is sent to the THP Service Module with an error status. Otherwise, the request is linked into the to_user queue.

IO_WRITE is called when a write I/O request is received. If the connection is closed or if the maximum number of writes has been exceeded, an I/O reply is sent to the THP Service Module with an error status. Otherwise, the request is put on the to_net queue.

IO_SPCL is called when an I/O special request is received from the THP Service Module. If it is a status request, the THP Service Module request is used to create an I/O special request to be sent to the TCP Module. If the special request asks that a specific THP control be sent, the control is linked into the to_net queue and an I/O special reply is sent to the THP Service Module.



THP SUBROUTINE CALLING HIERARCHY

* subroutine groups

To User. The TO_USER subroutine processes the to_user queue. The to_user queue contains user data or THP controls. Entries in the queue are processed until the queue is emptied or until queue element processing must be delayed. (Delay occurs when no user read buffers are available.)

If the entry contains user data, NET_DATA is called. NET_DATA copies network data to a per-user line buffer. NET_DATA then calls GIVE_USER_DATA. GIVE_USER_DATA determines if the user is a terminal or the THP Service Module. If the user is an INFE terminal, the data is written to the terminal. If the user is an INFE process, the data is copied into any previously received read I/O request buffer(s). NET_DATA continues this process until all data is copied to a line buffer, or the line buffer is full. In either case NET_DATA returns to TO_USER.

If a to_user queue entry contains a THP option, NET_OPTION is called. NET_OPTION puts the option in a standard form. NET_OPTION then calls an option specific handling subroutine. These subroutines perform negotiation based on the particular option. The GEN_OPT subroutine is called when a reply must be sent to the negotiation. GEN_OPT formats a network control message and links it into the to_net queue. TO_USER may call GIVE_USER_DATA directly if data was left in the per-user line buffer by a previous invocation of TO_USER.

To Net. The TO_NET subroutine processes the to_net queue. The to_net queue contains user data or THP controls. Entries in the queue are processed until the queue empties or until processing of one element must be delayed; usually because

no network transmit buffers are available. If the entry contains user data, THP_DATA_MSG is called. If the entry contains a THP control message, THP_CTRL_MSG is called. If a complete write I/O request was processed, IO_RPLY is called to send an I/O reply to the THP Service Module.

THP_DATA_MSG builds a THP data record in the current output buffer. If the data record fills the buffer, NEW_XMIT is called to send the buffer to the TCP Module and to locate a new buffer. If a new buffer was not available, the copying is terminated until a new buffer is available. If all of the data was copied and end-of-letter was set, NEW_XMIT is called to write the data to the TCP Module.

THP_CTRL_MSG formats a THP control message in the current output buffer. If there is not enough room in the buffer for the message, NEW_XMIT is called.

Remote Controlled Transmission and Echoing

Terminal data is normally echoed by the UNIX Terminal Handler. However, when a remote THP requests the RCTE (remote controlled transmission and echoing) option, the THP Module must perform echoing functions. When RCTE is in effect, terminal characters are copied into a local echo buffer before being linked into the to_net queue. When these characters are to be echoed (echoing is initiated by an RCTE control), characters are copied from the echo buffer into the terminal line buffer and written to the terminal.

Host-Terminal Communications

Connections between INFE terminals and the host are supported by the THP Module. This facility iis provided without the use of the network by tying together two Connection Management Blocks. One CMB simulates a connection between the host and the network. The other CMB simulates a connection between the INFE terminal and the network. Both the host and the INFE terminal appear to have a standard network connection.

The CMB's are tied together by inserting in each a pointer to the other. Data from each source is inserted into the appropriate to_net queue as in the standard case. When the TO-NET routine is invoked, it notes that this is a special case. Instead of writing the data to the net, it shunts the data from the to_net queue to the to_user queue of the other CMB. The data is then passed to the recipient in the usual way.

THP DATA STRUCTURES

The format of a THP Module Connection Management Block (CMB) follows. A Connection Management Block is allocated for each THP connection. Two Connection Management Blocks are allocated for each connection between an INFE terminal and the host.

```
struct cmb
{
    struct cmb *c_cmb_link;      /* used to link together cmb's */
    int     c_fid;               /* TCP file id */
    int     c_userid;            /* user process id */

    struct q_ele   *c_to_user;   /* user data & ctl queue */
    struct q_ele   *c_to_net;    /* net data & ctl queue */
    struct q_ele   *c_net_bufs;  /* free net write buffs */
    struct q_ele   *c_uread;     /* user read descriptors */
    char    c_nreads;            /* num of c_ureads */
    char    c_nwrites;           /* outstanding writes from user */
    char    c_nstty;             /* number of stty's outstanding */
    char    c_nctrl;             /* control requests out */
    struct q_ele   *c_sv_stty;   /* user sttys */
    struct q_ele   *c_sv_ctrl;   /* user ctl requests */
    int     c_t_mode[3];         /* current tty mode */
    int     c_rs_in;             /* approx rec size for input */
    int     c_rs_out;            /* approx rec size for output */
    char    c_stflags;           /* various status flags.  */
    char    c_nsynch;            /* outstanding synchs */
    char    c_nosynch;           /* synchs in output stream */
    char    c_naborts;           /* aborts on output */
    char    c_state;             /* the state of the connection. */
    char    c_sigflag;           /* required activity bits */
    char    c_icurs;             /* pos of the cursor on input */
    char    c_ocurs;             /* pos of the cursor on output */
    char    c_op_flags[20];      /* optn negotiation state */
    char    c_nnegout;           /* unreplied optn negotiations */
    char    c_nnegin;            /* rcvd option negotiations */
    char    c_ircte;             /* rcte echoing flags */
    char    c_orcte;             /* local rcte echoing flags */
    int     c_br_in;             /* break classes from remote thp */
    int     c_ixmit;             /* trans classes from remote thp */
    int     c_br_out;            /* break classes we sent them */
    int     c_oxmit;             /* trans classes we sent them */
    char    c_in_lw;             /* input line width */
    char    c_out_lw;            /* output line width */
    char    c_in_ps;             /* input page size */
    char    c_out_ps;            /* output page size */
    char    c_in_vtabs[NUM_TABS];/* input vertical tabstops */
    char    c_out_vtabs[NUM_TABS];/* output vertical tabstops */
```

```
    char    c_in_htabs[NUM_TABS];/* input horiz. tabstops */
    char    c_out_htabs[NUM_TABS];/* output horiz. tabstops */
    char    c_idp_cr;           /* input cr disposition */
    char    c_odp_cr;           /* output cr disposition */
    char    c_idp_lf;           /* input lf disposition */
    char    c_odp_lf;           /* output lf disposition */
    char    c_idp_ff;           /* input ff disposition */
    char    c_odp_ff;           /* output ff disposition */
    char    c_idp_ht;           /* input ht disposition */
    char    c_odp_ht;           /* output ht disposition */
    char    c_idp_vt;           /* input vt disposition */
    char    c_odp_vt;           /* output vt disposition */
    char    c_tbuf[6];          /* temp record header locs */
    char    c_tbuf_p;           /* next c_tbuf slot */
    char    c_h_partial;        /* required header chars */
    int     c_part_count;       /* bytes left in this record */
    char   *c_tptr;             /* temp record storage */
    int     c_tlen;             /* length of the above buffer */
    int     c_t_flags;          /* tty interface flags */
    char    c_tty_fid;          /* tty file id */
    char    c_com_flag;         /* command flag character */
    char    c_eline;            /* end-line character */
    char    c_erase;            /* the erase character */
    char    c_delete;           /* the delete character */
    char    c_interrupt;        /* the interrupt character */
    char    c_ao;               /* the abort output character */
    char    c_escape;           /* the literal escape character */
    char   *c_pbuf;             /* pointer to line buffer */
    char   *c_nx_pbuf;          /* next pbuf character */
    char   *c_fr_pbuf;          /* Next free character in pbuf */
    char   *c_e_pbuf;           /* addr of curr end of line buff */
    char   *c_ibuf;             /* tty read buffer */
    char   *c_e_ibuf;           /* the end of the c_ibuf */
    char   *c_fr_ibuf;          /* next char position */
    char    c_rbuf[RBUF_SIZE];  /* RCTE echo storage */
    char    c_nx_rbuf;          /* nxt c_rbuf char to echo */
    char    c_fr_rbuf;          /* past last echo char */
};
```

THP SUBROUTINES

The following is a list of the major THP subroutines.

cls_req
: Handles a close io message from the service.

cls_rply
: Handles a close reply from the TCP Module or a user tty.

io_read
: Handles a read request from the service.

io_write
: Handles a write request from the service.

ior_read
: Handles a read reply from the TCP Module or a user tty.

ior_write
: Handles a write reply from the TCP Module or a user tty.

net_data
: Processes data elements from the to-user queue. It copies data into the line buffer from where it is passed to the service or tty.

net_option
: Processes option elements from the to-user queue. It checks the option for validity, makes changes to the current state of options, and initiates any appropriate response options. Much of the work on individual options are done by external filter routines, most of which have not been written.

opn_req
: Handles an open request from the service.

opn-rply
: Handles an open reply from the TCP Module or a user's tty.

scan_rcte
: When the RCTE (Remote Controlled Echoing) option is in effect, this routine is called to handle echoing of data typed by the user.

tcp_rdrply
: This routine (together with routines it calls) scans buffers from the net and puts appropriate data and control messages into the to-user queue.

thp
: This is the main routine. It performs initialization, forwards io messages to the appropriate routines for processing, and calls to-user and to-net when appropriate.

thp_ctrl_msg
: Formats a THP control record into the current transmit buffer.

thp_data_msg
: Formats a THP data record into the current transmit buffer.

thp_to_internal
: This routine scans data received from the net for CR-LF sequences, and puts data messages into the to-user queue.

to_net
: This is the routine that reads the to-net queue. It calls thp_ctrl_msg and thp_data_msg to format transmit buffers, and writes them to the TCP Module.

to_user
: This routine reads the to-user queue. Options are checked for validity and responses generated. Data is passed to the user.

tty_data
: This routine actually looks at data typed on a user tty. The routine checks for special characters, and calls got_line when a line has been typed.

tty_rdrply
: This handles a read reply from a user tty. Data is passed to tty_data.

APPENDIX I: TCP Subroutine Specifications

TCP CALLING HIERARCHY

The following is a subroutine calling hierarchy for the TCP Module. This list contains all of the subroutines in the TCP Module.

abort	calls	cancel, mvq, reply, rlse_rcv, vdeq
ack	calls	cancel, diff, log, mvq, panic, sndreply
assemble	calls	diff, insert, last_buf, log, sadd, to_user
b_ack	calls	checksum, get_buf, panic, send, venq
bld_assm	calls	deq, iomap, log, panic
blpkt	calls	checksum, d_opt_lg, diff, enable, get_buf, panic, window, log, mv_data
bye	calls	panic
checksum	calls	cksum
cksum	calls	onecadd
combine	calls	enq, merge
d_ack_att	calls	diff
d_options	calls	log, printf
d_seq_att	calls	diff, log, panic, sadd
enable	calls	activate, panic, time
free_buf	calls	panic, venq
get_buf	calls	vdeq
getc	calls	iomap, panic
init	calls	enq, getpid, ioaddr, open, p_open, panic, signal, venq
init_read	calls	free_buf, get_buf, log, p_read, venq
insert	calls	bld_assm, combine, panic, sv_buf
isn	calls	window
last_buf	calls	mvq, panic, rdreply
log	calls	ctime, printf, time
log_all	calls	log_gbl, log_qs, log_tcb

log_gbl	calls	printf
log_opn	calls	printf
log_pkt	calls	printf
log_qs	calls	printf
log_tcb	calls	printf
main	calls	exit, free_buf, init, iowait, log, netin, p_openr, panic, send, u_event, user_in, vdeq
merge	calls	enq, panic
mv_data	calls	diff, iomap, panic, sadd
mvq	calls	deq, enq
n_closed	calls	b_ack, log
n_closing	calls	abort, ack
n_cls_wt	calls	ack, assemble, panic, reset, send
n_estab	calls	ack, assemble, printf, reset, rlse_rcv, sadd, send
n_fin_wait	calls	abort, ack, assemble, log, panic, reset, send
n_listen	calls	assemble, b_ack, isn, log, send
n_syn_rcvd	calls	assemble, b_ack, log, opreply, reset, send
n_syn_sent	calls	ack, assemble, b_ack, opreply, reset, send
netin	calls	(unknown), checksum, d_ack_att, d_opt_lg, d_seq_att, init_read, log, net_security, p_scan, panic, vdeq, d_options, net_tcb
num	calls	getc
oparam	calls	any, getc, log, match, num
opreply	calls	ioreply, vdeq
panic	calls	iot, log, log_all, printf
rdreply	calls	ioreply
reply	calls	ioreply
reset	calls	abort, isn, panic, send

rlse_rcv	calls	mvq, reply
send	calls	bld_pkt, to_net, vdeq
shuffle	calls	mvq
sndreply	calls	ioreply
sv_buf	calls	panic, venq
tevent	calls	activate, time, timeout
timeout	calls	abort, send
to_net	calls	p_write
to_user	calls	diff, mvq, panic, rdreply, sadd, shuffle
u_abort	calls	abort, panic, reply, send
u_close	calls	abort, panic, reply, send
u_read	calls	mvq, netin, panic, reply
u_send	calls	mvq, panic, reply, sadd, send
u_special	calls	panic, reply
u_status	calls	iomap, panic, reply
u_stty	calls	reply
user_in	calls	(unknown), log, oparam, open_itt, opreply, reply, u_security, venq, panic
vdeq	calls	mvq
venq	calls	mvq, panic

TCP SUBROUTINES

The following is a description of each of the major subroutines in the TCP Module.

```
NAME:
        ack

FUNCTION:
        see what has been acked and process it

ALGORITHM:
        if ack attribute in not ok
                return
        get the difference between the packet seq and the send seq

        copy the packet sequence into the send sequence
        copy the window from the packet

        if a SYN was acked
                reset the SYN bit
                decrement the number of sequence numbers acked

        if a FIN was acked
                reset the FIN bit
                set FIN_ACKED
                decrement the number of sequence numbers acked

        if sequence number acked is zero
                return

        if it is negative
                log it
                return

        if the URGNET was acked
                reset the send urgent bit

        for ever -- release acked buffers
                if the send q is empty
                        panic

                point at the first one
                decrement seq_nums by the segments logical length

                if seq_nums >= 0 (acked the whole thing)
                        increment bytes acked by its physical length
                        move the buffer descriptor to the free queue
                        generate a reply for the buffer
                else    partial ack
                        add the logical length to seq_nums
                        add seq_nums to physical bytes acked
                        if seq_nums is zero
                                break

                        update the send buf descriptor --
                                reset the EOL bit in flags
                                increment offset by seq_nums
                                decrement log_len by seq_nums
                                decrement physical length by seq_nums
```

```
                        if physical length is less than zero
                                panic
                        break
        decrement out by number physical bytes acked
        decrement number q'd to go by physical bytes acked

PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        panic
        sndreply
        mvq

CALLED BY:
        n_syn_sent
        n_estab
        n_fin_wait
        n_cls_wt
        n_closing

HISTORY:
        initial coding 3/8/78 by David Healy of DTI
```

until processing of one element must be delayed; usually because


```
NAME:
        assemble

FUNCTION:
        first step in giving data to the user

ALGORITHM:
        if seq_att is DUP
                return

        switch on tp -> t_state
        CLOSED
                return

        case LISTEN
        case SYN_SENT
                if seq_att is not IN_SEQ
                        break;
                if we got a syn
                        increment a rcv sequence number
                break;

        case SYN_RCVD
                break;

        case CLOSE_WAIT
        case CLOSING
                if pkt seq is > rcv seq
                        log data after FIN
                break

        case ESTABLISHED
                insert
                to_user

                if there is a fin and all data has been give
                to the user
                        mark FIN_RCVD
                break

PARAMETERS:
        a tcb ptr
        a packet ptr

RETURNS:
        nothing

GLOBALS:
        seq_att

CALLS:
        diff
        log
        insert
        to_user
        last_buf
```

CALLED BY:

n_listen
n_syn_sent
n_syn_rcvd
n_estab
n_fin_wait
n_cls_wait

HISTORY:

initial coding 3/8/78 by David Healy of DTI

```
NAME:
        checksum

FUNCTION:
        checksum a packet

ALGORITHM:
        checksum the internet header
        checksum the tcp header
        if there are any options
                while there is option data
                        compute number of checksumed bytes
                        checksum them
                        skip over unchecksumed option bytes
        if there is any data left in the packet
                checksum it
        if the checksum is not zero
                ones compliment it

PARAMETERS:
        ainp    a packet ptr

RETURNS:
        checksum

GLOBALS:
        none


CALLS:
        cksum

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/77 by David Healy of DTI
```

```
NAME:
        d_ack_att

FUNCTION:
        determine ack attributes of the packet --
                NONE -- no ack present bit
                OK   -- between snd seq and nxt seq
                DUP  -- not OK
        determine number seq nums acked

ALGORITHM:
        if there is no ack bit
                ack_att = NONE
        acked = packet ack - snd seq
        if acked > 0 and if
           packet ack <= nxt seq
                ack_att = OK
        else
                ack_att = DUP

PARAMETERS:
        tcp ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        acked
        ack_att

CALLS:
        diff

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy of DTI
```

```
NAME:
        d_seq_att

FUNCTION:
        determine the attributes of the packet sequence --
                IN_SEQ -- packet seq is the same as rcv seq
                IN_WIN -- some part of the packet is in our window
                DUP -- none of the above
        if part of the sequence numbers are duplicate
                adjust packet seq to make it look in sequence
                adjust data_lg and data_off
                reset duplicate controls
                adjust the urgent offset number
                reset the urgent if its now negative
        set dup_num to the amount the packet is out of sequence

ALGORITHM:
        compute the difference between the rcv seq and pkt seq
        if it is zero
                seq_att = IN_SEQ
        else
                if the window edge is >= pkt seq and if
                   the pkt edge is >= rcv seq
                        if duplicate stuff
                                seq_att = IN_SEQ -- it will be
                                adjust pkt seq
                                adjust pkt urg offset
                                if negative reset pkt urg bit
                                if pkt SYN
                                        reset it
                                        dec ctrl lg and num dup
                                if num dup still > 0
                                        reset pkt BOL
                                decrement data_lg by num duplicate
                                increment data_off by num duplicate
                                num duplicate = 0
                        else
                                seq_add = IN_WINDOW
                else
                        seq_att = DUP
        dup_num = num duplicates

PARAMETERS:
        tcb ptr
        pkt ptr

RETURNS:
        nothing

GLOBALS:
        seq_att
        data_lg
        data_off
        ctrl_lg
        dup_num
```

CALLS:

diff
log
panic

CALLED BY:

netin

HISTORY:

initial coding 3/6/78 by David Healy of DTI

```
NAME:
        d_options

FUNCTION:
        process the options in the tcp header

ALGORITHM:
        if there aren't any return

        point at the beginning of the options

        for ever
                get the next option byte

                switch on the option
                CASE BUFSIZ
                        copy the buffer size into the tcb
                        increment option byte counter
                        point at the next option
                        continue

                CASE O_END
                        increment the option byte counter
                        return the number of option bytes

                CASE O_PAD
                        increment the option byte counter
                        continue

                DEFAULT
                        get the option length
                        increment the option byte counter
                        point at the next option
                        continue

PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        number of option bytes

GLOBALS:

CALLS:
        log

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/77 by David Healy of DTI
```

```
NAME:
        init

FUNCTION:
        initialize queues and open the net and daemon file

ALGORITHM:
        init the following queues
                FR_IDQ
                FR_RCVQ
                FR_SNDQ
                FR_BUFQ
                FR_ASSMQ

        get a ioport
        p_open to open the net
        open the tcp daemon file

PARAMETERS:
        none

RETURNS:
        nothing

GLOBALS:
        idqs
        rcvqs
        sndqs
        assms
        buf

        FR_RCVQ
        FR_SNDQ
        FR_IDQ
        FR_ASSMQ
        FR_BUFQ

        mypid
        segbase
        tcpdaemon


CALLS:
        enq
        getpid
        ioaddr
        p_open
        open
        panic

CALLED BY:
        main

HISTORY:
        initial coding 3/8/77 by David Healy of DTI
```

```
NAME:
        main

FUNCTION:
        call init to have things initialized
        provide the driving loop for the program

ALGORITHM:
        call init to have things initialized


        for ever
                wait for an io event
                call the appropriate routine


PARAMETERS:
        none

RETURNS:
        never

GLOBALS:
        event
        replyid

CALLS:
        init     initialize things
        iowait   wait for an io event
        u_event  handle events from users
        user_in  handle io requests from users
        p_openr  psip open response handler
        log      log stuff
        netin    handle network read completes
        mvq      move from one q to another

CALLED BY:
        no one

HISTORY:
        initial coding 3/8/77 by David Healy of DTI
```

```
NAME:
        netin

FUNCTION:
        first level handling of a network message

ALGORITHM:
        if a buffer was not specified
                get one from the net read queue
                decrement number of reads outstanding

        if p_scan says it is not for us to look at
                log that
        else
                if the checksum is not correct
                        log that
                else
                        compute
                                internet header length
                                tcp header length
                                number tcp option bytes
                                number of data bytes

                        set ptrs to the internet and tcp headers
                        find the tcb associated with the packet

                        if net_security fails
                                log that
                        else

                                count num seq consuming ctls
                                set seq len consumed (seq_lg) to
                                        num data bytes+ctls.

                                determine sequence and ack attributes
                                call state dependent code
        start up another network read

PARAMETERS:
        bufp or 0

RETURNS:
        nothing

GLOBALS:
        netreads        number of netword reads outstanding
        cur_buf         current buffer
        inp_hlg         internet header length
        pkt_hlg         tcp header length
        data_lg         number of data bytes
        cur_inp         pointer to internet header
        cur_pp          pointer to tcb header
        ctrl_lg         number to tcp controls that consume seq nums
        seq_lg          length of packet in sequence numbers
        data_off        offset from the tcb header to the data
```

CALLS:

vdeq
p_scan
checksum
net_tcb
net_security
d_opt_lg
d_seq_att
d_ack_att
(*n_func) ()
log

CALLED BY:

main

HISTORY:

initial coding 3/8/77 by David Healy of DTI

NAME:

net_security

FUNCTION:

screen incoming packets for correct security levels
and adjust the TCB send levels if appropriate.

if the incoming packet does not match an active TCB,
attempt to bind it to an incomplete one.

ALGORITHM:

```
If the tcb pointer is NULL,
        call PKTBIND and return whatever he returns

If packet TCC and TCB TCC don't match,
        return -1

If packet security level is out of send range,
        return -1

If packet precedence is out of range,
        set send precedence against the stops
else
        set send precedence to that of packet
```

PARAMETERS:

pktp	- pointer to packet
tcbp	- pointer to TCB packet came on

RETURNS:

0	- if everything matches ok
-1	- if the packet is bad

GLOBALS:

none

CALLS:

pktbind

CALLED BY:

HISTORY:

initial coding

```
NAME:
        n_closed

FUNCTION:
        handle a network packet with tcb in the closed state

ALGORITHM:
        if he sent us a reset
                log it
        else
                send him a believable reset

PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        log
        b_ack

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy of DTI
```

NAME:

n_closing

FUNCTION:

handle a packet when we are waiting for our fin to be acked

ALGORITHM:

```
if it is a duplicate packet
        return
see what he acked
if he acked our fin
        abort the connection
```

PARAMETERS:

tcb ptr
packet ptr

RETURNS:

nothing

GLOBALS:

none

CALLS:

ack
abort

CALLED BY:

netin

HISTORY:

initial coding 3/8/78 by David Healy of DTI

```
NAME:
        n_cls_wait

FUNCTION:
        handle a packet when we are in the close_wait state

ALGORITHM:
        if the packet length is non zero
                remember it
        if seq_att is DUP
                see what we should send
                return
        if no ack bit
                panic
        see what he acked
        if we got a reset
                reset us
                return
        see what he gave us
        send what we can

PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        send
        panic
        ack
        reset
        assemble

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy of DTI
```

```
NAME:
        n_estab

FUNCTION:
        handle a packet when we are in the established state

ALGORITHM:
        if the packet length is non zero
                remember it
        if the sequence attribute is DUP
                send what we can
                return
        if ack attribute is none
                log it
                return
        if ack attribute is ok
                see what he acked
        if we got a reset
                reset us
                return
        if we got an urgent
                remember the sequence number of the last urgent byte
                if we aren' currently in urgent
                        tell the user
                        mark it
        see what we got in the way of data

        if we got a fin in sequence
                release all rcv buffers
                state to closing
        see what we can send

PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        reset
        send
        ack
        assemble
        rlse_rcv

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy of DTI
```

```
NAME:
        n_fin_wait

FUNCTION:
        handle a packet when we are in the fin_wait state

ALGORITHM:
        if packet length is non zero
                remember it
        is seq_att is DUP
                see what we can send
                return
        if no ack bit
                panic
        if duplicate ack
                log it
        see what he acked

        if we got a reset
                reset us
                return

        if we got a FIN
                remember it

        process what he gave us

        if the fin has been given to the user
                if our fin hasn't been acked
                        state to CLOSING
                else
                        abort the connection
                        return
        see what we can send

PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        send
        log
        panic
        ack
        assemble
        reset
        abort

CALLED BY:
        netin
```

HISTORY:

initial coding 3/8/78 by David Healy of DTI

```
NAME:
        n_listen

FUNCTION:
        handle a network when we are in the listen state

ALGORITHM:
        if he sent us an ack (no good)
                if we got a reset
                        log it
                else
                        send a believable reset
                return

        if we got a reset
                log it
        else
                if we got a SYN          (good news)
                        copy packet sequence and window
                        select an isn
                        send a SYN/ACK
                        state to SYN_RCVD
                        process what we got


PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        log
        b_ack
        assemble
        send

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy
```

```
NAME:
        n_syn_rcvd

FUNCTION:
        handle a packet when we are in the syn_rcvd state

ALGORITHM:
        is the packet a duplicate
                did we get a reset
                        reset us
                else
                        did we get an ack
                                send a believable reset
                                reset us
                return

        if the ack attribute is not ok
                if we didn't get a reset
                        send a believable reset
                reset us
        else
                if we got a reset
                        reset us
                else
                        if we got a SYN -- weird indeed
                                log it
                        else
                                see what we got
                                state to ESTABLISHED
                                reply to any q'd opens
                                send an ACK

PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        reset
        b_ack
        assemble
        send
        opreply
        log

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy of DTI
```

```
NAME:
        n_syn_sent

FUNCTION:
        handle a packet when we are in the syn_sent state

ALGORITHM:
        did we get an ack
                if ack_att is not OK (BAD indeed)
                        send a believable reset
                        return
                if we got a reset (start over)
                        reset us
                        return
                else
                        see what he acked

        if we got a syn
                copy packet sequence into our rcv sequence
                copy window into our rcv window
                see what he acked
                if he didn't ack the fin (rats)
                        state to SYN_RCVD
                else
                        state to ESTABLISHED
                        reply to any q'd opens
                send an ack
        else
                send a believable reset
                reset us

PARAMETERS:
        tcb ptr
        packet ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        b_ack
        reset
        opreply
        assemble
        send
        ack

CALLED BY:
        netin

HISTORY:
        initial coding 3/6/78 by David Healy of DTI
```

```
NAME:
        oparam

FUNCTION:
        parse the open parameters

ALGORITHM:
        set op_offset and op_left to 0

        initialize the open parameter struct

        for ever
                skip over leading delimiters
                copy the next parm into a string
                see if it matches anything we know about
                if index >= 0 and if args[index].flags == expected
                        get the number

                switch on index
                        log bad parm
                        return -1

                case O_FPORT
                        copy the fgn port into open parm struct
                        continue

                case O_LPORT
                        copy the local port into open parm struct
                        continue

                case O_FTCP
                        copy the fgn tcb into open parm struct

                        continue

                case O_FNET
                        copy the fgn net id into open parm struct
                        continue

                case O_BUF
                        copy the buffer size into open parm struct
                        continue

                default
                        or the arg.flags into open parm struct
                        continue

PARAMETERS:
        none

RETURNS:
        -1 if bad parm
        1 if successful

GLOBALS:
        the open parameter struct
```

CALLS:

getc
any
match
num
log

CALLED BY:

userin

HISTORY:

initial coding 3/8/78 by David Healy of DTI

```
NAME:
        reset


FUNCTION:
        reset the tcb

ALGORITHM:
        switch on the tcb state

        case CLOSED
                return

        case LISTEN, SYN_SENT, SYN_RCVD
                if init wanted
                        select an isn
                        set the SYN control bit
                        send a packet
                        set the state to SYN_SENT
                else
                        set the state to LISTEN
        default
                abort the connection

PARAMETERS:
        tcb ptr

RETURNS:
        nothing

GLOBALS:
        none

CALLS:
        panic
        send

CALLED BY:
        n_syn_sent
        n_syn_rcvd
        n_estab
        n_close_wait
        n_fin_wait

HISTORY:
        initial coding 3/7/78 by David Healy of DTI
```

```
NAME:
        send

FUNCTION:
        decide what to send to the net

ALGORITHM:
        while number netwrite outstanding is less that max allowed
                if a tcb was specified
                        if all user data has been send and there
                                are not controls to send
                        reset tcb was specified

                if no tcb was specified
                        if something on the believable queue
                                if to_net successful
                                        continue
                                else
                                        break

                find the most deserving tcb
                if none
                        return

                if to_net (bld_pkt (the selected tcb)) fails
                        break
                reset tcb was specified


PARAMETERS:
        a tcb ptr or
        0

RETURNS:
        nothing

GLOBALS:
        netwrites
        BLVQ

CALLS:
        vdeq
        to_net
        bld_pkt

CALLED BY:
        main
        n_listen
        n_syn_sent
        n_syn_rcvd
        n_estab
        n_fin_wait
        n_cls_wait
        reset
        b_ack
        u_close
```

u_abort
u_send

HISTORY:
initial coding 3/8/78 by David Healy of DTI

```
NAME:
        to_net

FUNCTION:
        write a buffer to the net

ALGORITHM:
        point at the internet header
        p_write (bufp, packet length + local network pad)
        if successful
                increment netwrites
                return 1
        else
                return 0

PARAMETERS:
        buffer ptr

RETURNS:
        1 if successful
        0 if not

GLOBALS:
        netwrites

CALLS:
        p_write

CALLED BY:
        send

HISTORY:
        initial coding 3/7/78 by David Healy of DTI
```

```
NAME:
        to_user

FUNCTION:
        give the user completed buffers

ALGORITHM:
        thru the rcv buffers
                point at the frag descriptor
                if none
                        break;
                point at the oldest one
                if the fragment offset is zero and if
                        fragment length matches rcv buf length
                        or fragment EOL

                        copy the fragment length into the buff desc
                        do rubber-EOL stuff
                        free the fragment descriptor
                        rdreply
                        free the rcv buf descriptor
                else
                        if frag offset = 0
                                left = frag length
                                seq nums =+ frag length
                        break;

        if consumed sequence numbers
        increment the rcv seq by seq nums - already acked
        already acked = left

        if user was in urgent mode and he no longer is
                reset the urgent bit

PARAMETERS:
        tcb ptr

RETURNS:
        nothing

GLOBALS:
        FR_RCVQ

CALLS:
        shuffle
        panic
        rdreply
        mvq

CALLED BY:
        assemble

HISTORY:
        initial coding 3/6/78 by David Healy of DTI
```

```
NAME:
        user_in

FUNCTION:
        first level handling of a user request

ALGORITHM:
        if its an open
                get the open params
                open_it
                if get a tcb ptr back
                        q the open replyid id
                        if state is established
                                opreply
                else
                        reply with an error
                return

        check the daemon id over
        get a tcb ptr
        if the state is closed
                panic

        the the security is bad
                reply with an error
                log it
                return

        call the appropriate user routine

PARAMETERS:
        none

RETURNS:
        nothing

GLOBALS:
        tcbs
        event
        (*u_func) ()

CALLS:
        oparam
        open_it
        venq
        ioreply
        reply
        panic
        u_security
        log
        (*u_func) (tp)

CALLED BY:
        main

HISTORY:
```

initial coding 3/6/78 by David Healy of DTI

```
NAME:
        u_abort

FUNCTION:
        handle user request to abort a connection

ALGORITHM:
        switch on the state

        case CLOSED
                panic -- impossible
        case LISTEN
        case SYN_SENT
                abort WAS_RESET
                reply SUCCESS
                break;

        default
                decrement number open

        case SYN_RCVD
                send a reset
                abort WAS_RESET
                reply success

PARAMETERS:
        tcb ptr

RETURNS:
        nothing

GLOBALS:
        replyid

CALLS:
        abort
        send
        reply

CALLED BY:
        userin

HISTORY:
        initial coding 3/6/78 by David Healy of DTI
```

```
NAME:
        u_close

FUNCTION:
        handles a users close request

ALGORITHM:
        decrement the number of users that have it open
        if > 0
                reply SUCCESS
                return

        switch on the tcb state
        CASE CLOSED
                panic
        case LISTEN
        case SYN_SENT
                abort it
                reply success
                break

        case SYN_RCVD
        case CLOSE_WAIT
                send a fin
                set the state to CLOSING
                save the replyid

        case ESTAB
                send a fin
                set the state to FIN_WAIT
                save the replyid

        case FIN_WAIT
        case CLOSING
                panic   impossible

PARAMETERS:
        tcb ptr

RETURNS:
        nothing

GLOBALS:
        replyid

CALLS:
        reply
        abort
        panic

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy of DTI
```

```
NAME:
        u_read

FUNCTION:
        process a users read request

ALGORITHM:
        switch on the tcb state

        case CLOSED
                panic

        case LISTEN, SYN_SENT, SYN_RCVD
                reply NOT_OPEN

        case FIN_WAIT, CLOSE_WAIT, CLOSING
                reply DYING

        case ESTABLISHED
                if too many reads out
                        reply ERROR
                if can't get rcv buf descriptor
                        reply NO_RES
                increment number of reads out
                initialize the rcv buf descriptor --
                        length
                        offset
                        replyid
                if there is a q'd packet
                        call netin to process it

PARAMETERS:
        tcb ptr

RETURNS:
        nothing

GLOBALS:
        event
        replyid
        FR_RCVQ

CALLS:
        panic
        reply
        mvq
        netin

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy of DTI
```

```
NAME:
        u_send

FUNCTION:
        handle a user send buffer

ALGORITHM:
        switch on the tcb state

        case CLOSED
                panic

        case LISTEN, SYN_SENT, SYN_RCVD
                reply NOT_OPEN

        case FIN_WAIT, CLOSING
                reply NOT_OPEN

        case CLOSE_WAIT, ESTABLISHED

                if too many sends out
                        reply ERROR
                if can't get send descriptor
                        reply NO_RES
                increment number of sends out
                init the send descriptor --
                        length
                        offset
                        replyid
                        log_len
                        flags
                if currently not in letter
                        set the BOL bit

                and buffer length to fragment size and mod it buf size

                if user requested EOL
                        set EOL BIT
                        reset INLETTER
                        if fragment must add on for RUBBER-EOL
                                add in buffer size - fragment size
                        reset fragment size
                else
                        set INLETTER

                if user requested urgent
                        set the URG bit
                        compute seq num of last urgent octet

                send some stuff off

PARAMETERS:
        tcb ptr

RETURNS:
        nothing
```

GLOBALS:
        replyid
        FR_SNDQ
        event

CALLS:
        reply
        panic
        mvq
        sadd
        send

CALLED BY:
        netin

HISTORY:
        initial coding 3/8/78 by David Healy of DTI

APPENDIX II: THP Subroutine Specifications

THP CALLING HIERARCHY

The following is a subroutine calling hierarchy for the THP Module. This list contains all of the subroutines in the THP Module.

buf_addr	calls	mapin
cls_req	calls	iorply, q_to_net
cls_rply	calls	iorply, read_q, remove_cmb
flush_q	calls	free, iorply, read_q
gen_opt	calls	alloc, q_to_net
get_cmb	calls	alloc
give_user_data	calls	buf_addr, freeport, iorply, read_q, write
got_line	calls	parse_com, q_to_net, scan_rcte
ior_read	calls	tcp_rdrply, tty_rdrply
ior_write	calls	free, write_q
io_read	calls	iorply, q_to_net, write_q
io_write	calls	iorply, q_to_net
mapin	calls	iomap
net_data	calls	buf_addr, give_user_data, rewrite_q, unmap
net_option	calls	(unknown), buf_addr, gen_opt, rewrite_q, same_param, stty_reply
new_xmit	calls	read_q, write
opn_req	calls	alloc, freeport, get_cmb, iorply, mapin, nbopen, remove_cmb, write_q
opn_rply	calls	alloc, close, free, iorply, read, read_q, remove_cmb, write_q
parse_com	calls	(unknown), eq_init, printf, scan_word
partial_scan	calls	alloc, proc_rec
proc_rec	calls	q_to_net, q_to_user, start_urgent, thp_to_internal
q_ctrl	calls	q_to_net
q_to_net	calls	write_q

q_to_user	calls	write_q
read_q	calls	deq, enq
remove_cmb	calls	flush_q, free, iorply, read_q
rewrite_q	calls	deq
same_param	calls	lex_comp
scan_rcte	calls	printf, putc
tcp_rdrply	calls	free, partial_scan, proc_rec, q_to_user, read, start_urgent, thp_to_internal
thp	calls	(unknown), ioaddr, iorply, iowait, to_net, to_user, write
thp_ctrl_msg	calls	new_xmit, new_xmit
thp_to_internal	calls	q_to_net, q_to_user
to_net	calls	buf_addr, close, flush_q, free, freeport, iorply, new_xmit, read_q, rewrite_q, thp_ctrl_msg, thp_data_msg, unmap
to_user	calls	flush_q, free, give_user_data, kill, net_data, net_option, read, read_q, rewrite_q, scan_rcte
tty_data	calls	got_line, printf, q_ctrl, q_to_net
tty_rdrply	calls	alloc, got_line, mustfail, q_to_net, read, tty_data
write_q	calls	deq, enq

THP SUBROUTINES

The following is a description of the major THP subroutines.

NAME:
cls_req

FUNCTION:
Handles a close request from a user process.

ALGORITHM:
If the connection is in the state where the user already closed a not-yet-completed connection, complete the close with an error.
Otherwise, put a close request into the to-net queue.
The connection will be closed after all data has been written.

PARAMETERS:
reply_id integer The reply id for the request

RETURNS:
Nothing

GLOBALS:
cp
qe
thpios

In the current CMB:
c_state

CALLS:
iorply (sys)

CALLED BY:
thp (main)

HISTORY:
17 Feb 1978 D A Willcox of DTI Initial coding

```
NAME:
        cls_rply

FUNCTION:
        Handles a close reply io message.  This can be either
        from the tcp or from a user terminal.

ALGORITHM:
        If it is a close for a user tty:
            If there is no network connection
                Remove the CMB
            else
                Fix things up so that when the close comes from
                the net (it will), the CMB will be removed.
        otherwise
            If we are in a state where no reply has been sent
                to the user's close:
                Send a reply to the user's close.
            Remove the cmb.

PARAMETERS:
        None

RETURNS:
        Nothing

GLOBALS:
        cp
        qe =
        thpios

        In the current CMB:
        c_fid
        c_state =
        c_stflags =
        c_to_net

CALLS:
        iorply (sys)
        read_q
        remove_cmb

CALLED BY:
        thp (main)

HISTORY:
        17 Feb 1978     D A Willcox of DTI      Initial coding
```

```
NAME:
        io_read

FUNCTION:
        Handles a read request io message.

ALGORITHM:
        If the TCP has reported the connection to be closed
            Reject the request.
            Return

        If the number of allowable reads has been exceeded
            Reject the request.
            Return

        If this is a read coming in when there were none
            outstanding before:
            Put an EOM message into the to-net queue.
                This is for use when GA's are being generated.

        Save up a descriptor for the read buffer.
        Set flag indicating that to_user should be called.

PARAMETERS:
        reply   int     The reply id for the read.

RETURNS:
        Nothing

GLOBALS:
        cp
        qe =
        thpios =

        In the current cmb:
        c_nreads =
        c_ureads =
        c_sigflags =

CALLS:
        iorply (sys)
        q_to_net
        write_q

CALLED BY:
        thp (main) thru iom_rtn

HISTORY:
        14 Mar 1978     D A Willcox of DTI      Initial coding
```

```
NAME:
        io_write

FUNCTION:
        Processed a write request io message to the thp.

ALGORITHM:
        If output should be rejected
            Reject the write
            return

        If the max number of allowable writes has been exceeded
            Reject the write
        else
            Put a data message into to-net queue.
            Put in a message to reply to the write.
            If there are reads outstanding
                Put in a message indicating EOM, for use when
                    sending GA's.

PARAMETERS:
        reply   int     The reply id for the write request.

RETURNS:
        Nothing

GLOBALS:
        cp
        qe =
        thpios  The global io message

        In the current cmb:
        c_nreads
        c_nwrites
        c_state

CALLS:
        ioreply (sys)
        q_to_net

CALLED BY:
        thp (main) thru iom_rtn

HISTORY:
        14 Mar 1978     D A Willcox of DTI      Initial coding
```

NAME:

ior_read

FUNCTION:

Handle a read reply io message.

ALGORITHM:

```
If it is a rply from a read to a tty:
    Call tty_rdrply.
else
    Call tcp_rdrply.
```

PARAMETERS:

None

RETURNS:

Nothing

GLOBALS:

cp
thpios

In the current cmb:
c_tty_fid

CALLS:

tcp_rdrply
tty_rdrply

CALLED BY:

thp (main) thru iom_rtn

HISTORY:

14 Mar 1978 D A Willcox of DTI Initial coding

```
NAME:
        ior_write

FUNCTION:
        Handle a write request io message.

ALGORITHM:
        If it is a reply from a write to a tty:
            Advance the pointer to indicate what was written from
                the buffer.
            Set flag that it is now OK to write to the tty again.
        else
            If there was an error in the write:
                Free the transmit buffer.
                Change conn state to indicate closing conn.
            else
                Save up the buffer for later use.
                Indicate that to-net processor should be called.

PARAMETERS:
        None

RETURNS:
        Nothing

GLOBALS:
        cp
        qe =
        thpios =

        In the current cmb:
        c_net_bufs =
        c_nx_pbuf =
        c_t_flags =
        c_sigflag =
        c_state =

CALLS:
        free
        write_q

CALLED BY:
        thp (main) thru iom_rtn

HISTORY:
        14 Mar 1976     D A Willcox of DTI      Initial coding
```

NAME:

net_data

FUNCTION:

Net_data processes data elements in the to-user queue. It takes data from a network buffer and copies it into the printf buffer.
In the tty interface, data is written from the printf buffer directly to the tty. In an interface to the user process, a line is buffered in this buffer before being passed to the process to allow for local line editing.

ALGORITHM:

Fairly straightforward. Copy data from in to out (doing any translation necessary). If the output buffer fills up, generate a read response (for a process interface). If all of the data in the to-user queue element is not used, put an element back in the start of the queue to cause the remaining data to be copied later.

PARAMETERS:

qe — Pointer to a q_ele structure. This describes a hunk of network input data to give to the user.

RETURNS:

0 — All of the data in the queue element was processed.
-1 — The data in the element was not completely used.

GLOBALS:

cp — The pointer to the current connection management block

In the current cmb:
c_nreads
c_nsynch
c_to_user =
c_t_flags =
c_ureads =

CALLS:

buf_addr
give_user_data
read_q
rewrite_q

CALLED BY:

to_user

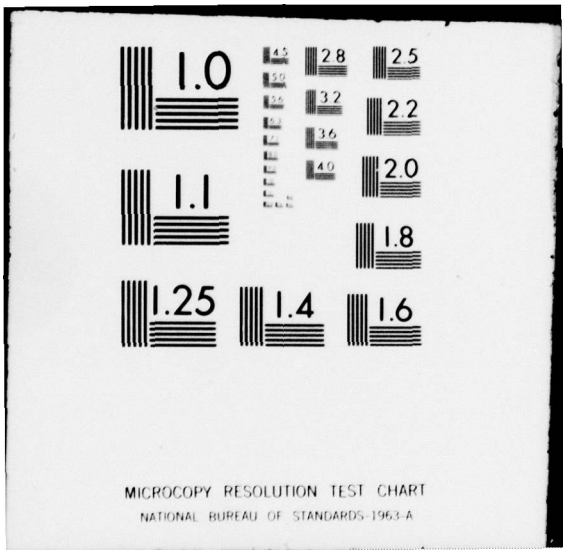
HISTORY:

6 Feb 1978 D A Willcox of DTI Initial Coding

AD-A060 668 DIGITAL TECHNOLOGY INC CHAMPAIGN IL F/G 9/2
INFE PROGRAM DESIGN SPECIFICATIONS. PHASE B. NFE SOFTWARE PROGR--ETC(U)
MAY 78 S F HOLMGREN, D C HEALY, D A WILLCOX
UNCLASSIFIED DTI-6 SBIE-AD-E100 082 NL

2 OF 2
AD
A060668

END
DATE
FILMED
01-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAME:

net_option

FUNCTION:

This routine handles option negotiations received from the net.

ALGORITHM:

There are a large number of possible cases which are handled here in a fairly brute-force manner. The first action is to copy the pieces of the option into local variables and set flags indicating whether the option is will/wont/do/dont, has a sender/receiver parameter, etc.

The semantics of the various options are handled by extern filter. The option-to-filter routine mapping is done with the table op_filter[]. These routines determine request is reasonable, and in some cases initiate a counter-negotiation.

PARAMETERS:

qe — pointer to q_ele structure
The queue element for the option that we read from the to-user queue. If the option cannot be handled right now, this element will be put back into the start of c_to_user.

RETURNS:

0 — If the element was handled ok.

-1 — If this option cannot be handled immediately. If there is insufficient buffer space to generate a reply. wThe queue element has returned to the queue.

GLOBALS:

cp
op_filter
op_flags

In the current cmb:
c_op_flags
c_to_user

CALLS:

buf_addr
gen_opt
rewrite_q
same_param

CALLED BY:

to_user

HISTORY:

14 Feb 1978 D A Willcox of DTI Initial coding

NAME:

opn_req

FUNCTION:

This routine processes an open request io message. Such a message arrives when a user process requests that a conn be opened.

ALGORITHM:

First, allocate a CMB for the connection. If there are no available CMB's, reject the open request.

If the allocation is successful, initialize the variables in the new cmb, and pass the open request on to the TCP. While open is pending, a descriptor for the open is stored in the to-net queue.

PARAMETERS:

reply_id integer The reply id for the user's request

RETURNS:

Nothing

GLOBALS:

cp =
qe =
thpios =

In the newly allocated cmb:
c_fid =
c_to_net =
c_to_user =

CALLS:

alloc
freeport
get_cmb
iorply (sys)
mapin
nbopen (sys)
write_q

CALLED BY:

thp (main)

HISTORY:

16 Feb 1978 D A Willcox of DTI Initial coding

```
NAME:
        opn_rply

FUNCTION:
        Processes an open reply message.  This can be either from the
        CP when a conn is established (or the open fails), or from
        the tty controller when we open a user's tty.

ALGORITHM:
        If it is an open reply from a user tty, allocate the printf
        and input buffers and initiate the first tty read.

        For an open reply from the TCP:
            Allocate and queue transmit buffers.
            Allocate receive buffers and issue reads to TCP.
            In addition, if this is an interface to a process:
                Allocate a to-user line buffer.
                Initialize editing characters, if they are needed.
                Reply to the user's open.

PARAMETERS:
        None

RETURNS:
        Nothing

GLOBALS:
        cp
        qe =
        thpios =

        In the current CMB:
        c_fid
        c_fr_pbuf =
        c_nx_pbuf =
        c_pbuf =
        c_state =
        c_stflags
        c_net_bufs =

CALLS:
        alloc
        close (sys)
        init_cmb
        iorply (sys)
        read (sys)
        read_q
        remove_cmb

CALLED BY:
        thp (main)

HISTORY:
        16 Feb 1976     D A Willcox of DTI      Initial coding
```

NAME:

scan_rcte

FUNCTION:

When the remote thp has WILL RCTE enabled, this routine scans the echo character buffer (cp->c_rbuf) for break chars. If an rcte set breaks has come in and no break character has been echoed, this routine echos data up to the break (assuming that echoing was specified in the set breaks command.)

ALGORITHM:

If we are in a state where we should echo characters, scan the saved tty input for a break. Echo any characters that should be echoed. If a break character was found in the buffer, only echo to there.

PARAMETERS:

None

RETURNS:

Nothing - works with global variables

GLOBALS:

ch_class
cp

In the current cmb:
c_fr_rbuf
c_nx_rbuf =
c_op_flags
c_rbuf
c_t_flags
c_to_net

CALLS:

printf (sys)
putc

CALLED BY:

got_line
to_user

HISTORY:

14 Feb 1978 D A Willcox of DTI Initial coding

NAME:

tcp_rdrply

FUNCTION:

This routine is called when data is received from the net.
It, and routines it calls, scan the input for data and controls
and put appropriate elements into the to-user queue.

ALGORITHM:

```
While there is still data in the input buffer:
    If there remains a partial header, call partial_scan
        to try to get a whole header.
    If there is a partial rec around (with a whole header)
    Put a data message into the to-net queue or copy a control
        message into an internal buffer.

    While there is still data in the input buffer:
        Process any STREAM data up to the next RM.
        Check if there is a whole record left in buffer.
        If not,
            set flags to get rest of record next time thru,
            do a break (get out of inner loop).
        Call proc_rec to handle the new record.

If, in the above, we sent any to - user queue elements that
    referred to data in this buffer, then send a to-user
    queue element to re-issue the tcp read.
Otherwise, re-issue the tcp read right now.
```

PARAMETERS:

ios	pointer to iostruct	Pointer to the reply message from the read.

RETURNS:

Nothing

GLOBALS:

```
cp
qe =

In the current CMB:
c_fid
c_h_partial =
c_sigflag =
c_tbuf
c_tlen =
c_to_net
c_to_user
```

CALLS:

```
partial_scan
proc_rec
q_to_user
read (sys)
start_urgent
thp_to_internal
```

CALLED BY:
thp (main)

HISTORY:
17 Feb 1978 D A Willcox of DTI Initial coding

NAME:

thp (main)

FUNCTION:

This is the main routine for the thp module.

ALGORITHM:

After some initial setup, go into an infinite loop doing iowaits.

When an io message for an established connection comes in, find the associated cmb for it. (Anything other than an open request from a user will be refused if there is no cmb assigned.)

Call a routine thru a table to handle the io message.

After handling the request, check if the request made it possible for us to send data to net and/or pass data to user. Keep doing this until we can do no more.

PARAMETERS:

None

RETURNS:

Never

GLOBALS:

cmb_tab
cp =
free_cmb =
iom_rtn
ptab
ptab_e =
used_cmb =

CALLS:

close (sys)
cls_req thru iom_rtn
cls_rply thru iom_rtn
io_gtty thru iom_rtn
io_read thru iom_rtn
io_special thru iom_rtn
io_write thru iom_rtn
ioaddr (sys)
iom_error thru iom_rtn
ior_gtty thru iom_rtn
ior_read thru iom_rtn
ior_special thru iom_rtn
ior_stty thru iom_rtn
iorply (sys)
iowait (sys)
opn_req thru iom_rtn
opn_rply thru iom_rtn
q_to_net
read_q

```
        server_ctrl
        to_net
        to_user
        tty_stty thru iom_rtn
        write (sys)

CALLED BY:
        No one

HISTORY:
        15 Feb 1978     D A Willcox of DTI      Initial coding
        14 Mar 1978     D A Willcox of DTI      Modified to call
```

NAME:

thp_ctrl_msg

FUNCTION:

Formats a thp control record into the current output buffer.

ALGORITHM:

Check if there is room for the record; if not, call new_xmit.

Build the output record in a straightforward manner.

PARAMETERS:

xbuf	ptr to q_ele struct	Descriptor of current output buffer
type	integer	Code for the type of message
first	ptr to character	First character of parameters (if any)
last	ptr to character	Last character of parameters for the control record

RETURNS:

-1	There is insufficient buffer space for the record.
0	The record has been put into the transmit buffer

GLOBALS:

None

CALLS:

new_xmit

CALLED BY:

to_net

HISTORY:

21 Feb 1978 D A Willcox of DTI Initial coding

NAME:

thp_data_msg

FUNCTION:

Formats a data record in current output buffer. If data record fills the buffer, dump the buffer to the TCP module.

ALGORITHM:

```
While there is still input data to write:
    If in record mode and  not appending to an old data
        record:
        Set up a record header.
    Copy as much data into the output record as possible.
    If there was not enough room for all of the data,
        Write the old buffer and get a new one.
```

PARAMETERS:

xbuf	ptr to q_ele struct	This is the descriptor of the current output buffer
first	ptr to character	The first data character to write
inlen	integer	Number of data characters to write

RETURNS:

integer - The number of characters written. If this is less than inlen, then insufficient transmit buffer space.

GLOBALS:

cp

In the current CMB:
c_stflags

CALLS:

new_xmit

CALLED BY:

to_net

HISTORY:

21 Feb 1978 D A Willcox of DTI Initial coding

NAME:

thp_to_internal

FUNCTION:

This routine is called to scan net data for end-of-line (cr-lf) sequences. It writes data msgs into the to-user queue, interspersed with messages marking end-of-message.

ALGORITHM:

The c_t_flags byte in the CMB is used to remember that a CR was seen in one buffer, but there was no character after it. (The character immediately following a CR must be either a LF or a null.) Thus:

If the character in the previous buffer was a CR, check the first character in the current buffer. Send a newline character to the user, if appropriate.

Scan thru each character in the buffer:
    If the current character is a CR:
        If this is not the last character in this buffer:
            If the next character is a newline:
                Change the CR to a NL.
            Write everything to here to the user.
            If it was CR-LF, write an EOM message.
        else
            Set a flag to remember that the last character in this buffer was a CR.
            Break out of the loop.
    else
        Just skip by the character.

Write all data from the last write to the end of the buffer.

PARAMETERS:

first	ptr to character	Pointer to first character read in
last	ptr to character	Pointer to one past last character

RETURNS:

Nothing

GLOBALS:

cp
qe =

In the current CMB:
c_t_flags =

CALLS:

q_to_user

CALLED BY:

proc_rec
tcp_rdrply

HISTORY:

20 Feb 1978	D A Willcox of DTI	Initial coding

```
NAME:
        to_net

FUNCTION:
        Reads to-net queue and calls routines to store data and control
        records into the transmit buffer.

ALGORITHM:
        Loop until we run out of to-net queue elements or transmit
            buffers:
            Read an element from the to-net queue.
            Switch on the function code:
                Functions that require putting data or controls into
                    the  transmit buffer are handled by passing
                    info to either thp_ctrl_msg or thp_data_msg.
                Others are done locally.
        If an entire message worth of data was put into the output
            buffer,
            Make sure it has been written.

PARAMETERS:
        None

RETURNS:
        Nothing

GLOBALS:
        cp

        In the current CME:
        c_nosynch =
        c_sigflag =
        c_stflags =
        c_to_net =
        c_net_buis =

CALLS:
        buf_addr
        freeport
        iorply (sys)
        new_xmit
        read_q
        rewrite_q
        thp_ctrl_msg
        thp_data_msg
        unmap

CALLED BY:
        thp (main)

HISTORY:
        21 Feb 1978     D A Willcox of DTI      Initial coding
```

HISTORY:

20 Feb 1978	D A Willcox of DTI	Initial coding

NAME:

to_user

FUNCTION:

This is the routine that reads the to-user queue containing data and controls. The TCP interface scans input records and puts appropriate entries into this queue.

ALGORITHM:

Loop until it is time to stop:

Read one element from the queue cp -> c_to_user.

Switch on the function code.
Some functions are handled right away, others are passed on to external routines.
If the func cannot be handled immediately (e.g. it is a data msg and no outstanding read from the user), then put the element into queue and return.

PARAMETERS:

None

RETURNS:

Nothing

GLOBALS:

cp

In the current CMB:
c_br_in
c_eline
c_erase
c_ircte =
c_ixmit =
c_naborts =
c_nosynch =
c_nsynch =
c_stflags =
c_t_flags =
c_t_mode
c_to_user

CALLS:

give_user_data
net_data
net_option
read_q
kill (sys)
rewrite_q
scan_rcte

CALLED BY:

thp (main)

HISTORY:

10 Feb 1978	D A Willcox of DTI	Initial coding

```
NAME:
        tty_data

FUNCTION:
        Processes input from the user's tty

ALGORITHM:
        Scan thru the data, check each character for special controls:
            If it is literal escape, take the next character literally.
            If it is the end-line character:
                Call got_line to process data or command.
                Send an EOM command to to-net processor.
            If it is character erase:
                Delete a buffered character or send an THP EC.
            If it is line delete:
                Delete line locally if we can,
                Otherwise, send a THP EL.
            If it is an IP or AO character:
                Call got_line to process previous data.
                Put an appropriate control into the to-net queue.
            If it is first character in line and it is the command flag:
                Set the flag that we are in command mode.
            If none of the above, buffer the character.

        Finally, if we are in STREAM mode or RCTE is in effect:
            Call got_line to send any data not yet sent.

PARAMETERS:
        first   ptr to char     Pointer to the first char read in
        len     integer         Number of chars that were read in

RETURNS:
        ptr to char - Pointer to the next address to read into

GLOBALS:
        cp

        In the current CMB:
        c_com_flag
        c_delete
        c_eline
        c_erase
        c_escape
        c_fr_ibuf
        c_interrupt
        c_op_flags
        c_stflags =
        c_t_flags =

CALLS:
        got_line
        printf
        q_ctrl

CALLED BY:
        tty_rdrply
```

HISTORY:

14 Feb 1978 D A Willcox of DTI Initial Coding

NAME:

tty_rdrply

FUNCTION:

Handles a read reply io message from the user's tty.

ALGORITHM:

Pass the data read in to tty_data, which does the scanning for special characters, and normally sends data to the net.

If the current input buffer is completely full and there is too much data in it that still hasn't been sent to the net, then call got_line to force the data out. (A command line could, conceivably, be truncated.)

If the input buffer is full to the high water mark,
Alloc a new input buffer,
Copy data that hasn't been sent from the old to the new buffer, and
Put a free buffer msg into the to-net queue, so that the old buffer will be freed at the correct time.

Initiate a new tty read. If read returns data right away, loop to start and handle the new data.

PARAMETERS:

None

RETURNS:

Nothing

GLOBALS:

cp
qe =
thpios =

In the current cmb:
c_e_ibuf =
c_ibuf =
c_fr_ibuf =
c_sigflags =
c_to_net

CALLS:

alloc
q_to_net
read (sys)
tty_data

CALLED BY:

thp (main)

HISTORY:

15 Feb 1976 D A Willcox of DTI Initial coding

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER DTI Document Number 6	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) INFE PROGRAM DESIGN SPECIFICATIONS (Phase B NFE Software Program Design Specifications)		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER DTI Document Number 6
7. AUTHOR(s) Steven F. Holmgren David Healy David Willcox		8. CONTRACT OR GRANT NUMBER(s) DCA100-77-C-0069
9. PERFORMING ORGANIZATION NAME AND ADDRESS Digital Technology Incorporated 302 East John Street Champaign Illinois 61820		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DCA/Command and Control Technical Center ATTN: C420 11440 Isaac Newton Square, North Reston, VA 22090		12. REPORT DATE May 15, 1978 13. NUMBER OF PAGES 114
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release;
distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No restriction distribution.

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Network Frontend

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Digital Technology Incorporated is developing an IOC Network Frontend (INFE) to connect a WWMCCS H6000 to AUTODIN II. A DEC PDP-11/70 running under a modified Network UNIX system is being used for the INFE. This document contains the program design specifications for the AUTODIN II protocol software used in the INFE.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)